Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Erik Ferfolja

Mobilna aplikacija za opozarjanje kolesarjev na nevarnosti v prometu

diplomsko delo

na univerzitetnem študiju

Mentor: doc. dr. Peter Peer
Asistent: as. Bojan Klemenc

Ljubljana 2012

Rezultati diplomskega dela so intelektualna lastnina avtorja ter Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01829/2012

Datum: 03.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ERIK FERFOLJA**

Naslov: **MOBILNA APLIKACIJA ZA OPOZARJANJE KOLESARJEV NA NEVARNOSTI V PROMETU**

MOBILE APPLICATION FOR WARNING CYCLISTS ABOUT DANGERS IN TRAFFIC

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

S pomočjo knjižnice OpenCV razvijte mobilno aplikacijo za opozarjanje kolesarjev na nevarnosti v prometu. Osredotočite se na dogodka približevanja in prehitevanja. Najprej naredite pregled sorodnih del. Nato podajte teoretično podlago uporabljenih metod ter zasnovo rešitve, ki bo omogočala segmentacijo ceste ter opozarjala na približevanja in prehitevanja. Rešitev implementirajte na platformi iOS. Na koncu naredite analizo rezultatov, tudi na ciljni mobilni platformi.

Mentor:

doc. dr. Peter Peer

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Erik Ferfolja, z vpisno številko **63020038**, sem avtor diplomskega dela z naslovom:

Mobilna aplikacija za opozarjanje kolesarjev na nevarnosti v prometu

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Petra Peera in as. Bojana Klemenca;
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) in ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela;
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, 15. oktobra 2012 Podpis avtorja:

Hvala Katji za pomoč in podporo, Gregorju za posojanje iPada, Bojanu Klemencu za strokovno pomoč in vzpodbudo ter mentorju, dr. Petru Peeru.

Staršem.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Pregled področja **3**

3 Opis problema **9**

4 Uporabljene metode OpenCV in njihova teoretična podlaga **13**

4.1 threshold 14
4.2 floodFill 14
4.3 add 15
4.4 erode 15
4.5 Sobel 16
4.6 Canny 18
4.7 Hough 20
4.8 watershed 22
4.9 KalmanFilter 23
4.10 goodFeatureToTrack 27
4.11 cornerSubPix 29
4.12 calcOpticalFlowPyrLK 29

5 Razvoj postopka **33**
5.1 Segmentacija ceste in določitev njenih robov 34
5.2 Zaznavanje približevanja 39
5.3 Zaznavanje prehitevanja 43

6 Implementacija postopka: aplikacija eyeCycle **47**
6.1 Obdelava posnetkov 51
6.2 LiveMode 53
6.3 Skupna funkcionalnost – DebugMode 54

7 Analiza rezultatov **59**
7.1 Uspešnost 59
7.2 Počasnost delovanja 63
7.3 Poraba baterije 65

8 Sklepne ugotovitve **67**

Literatura **69**

Povzetek

Razvoj avtomobilov brez voznika je v zadnjih letih močno napredoval. Napredek se kaže v uspešni registraciji prvega avtonomnega vozila. Tehnologija, ki se uporablja pri razvoju teh avtomobilov, je robustna in dokazano zanesljiva. Kolesarji, ki ravno tako sodelujejo v prometu, pa napredka te tehnologije za zdaj še ne morejo koristiti. Skušali smo razviti opozorilni sistem za kolesarje s pomočjo knjižnice OpenCV. Naš glavni cilj je bil opozoriti kolesarje na nevarnost prehitevajočih in približujočih se vozil. Sistem smo razvili za uporabo na pametnih telefonih. Glavna omejitev je bila poraba baterije. Uspešnost postopka smo ocenjevali na testnih posnetkih. Naša končna rešitev sicer še ima pomanjkljivosti, vendar predstavlja dobro izhodišče za nadaljnji razvoj. V zaključku predlagamo možnosti za izboljšave.

Ključne besede: OpenCV, iPhone, iOS, segmentacija ceste, monokularni sistem, opozorilni sistem, kolesarjenje

Abstract

In the field of artificial intelligence, the development of driverless cars has advanced to the point where an autonomous car is issued a license. The technology implemented in these driverless cars is proven to be reliable and robust. Bicyclists on the other hand do not benefit from advancements in this technology yet. We tried to develop a warning system for bicyclists using OpenCV. Our main goal was to warn bicyclists of approaching and overtaking cars. The system was developed for smart phone platforms so we were constrained by the battery lifetime. Our final solution still has some flaws but it is a good starting point for future development, so we propose possible improvements.

Keywords: OpenCV, iPhone, iOS, road segmentation, monocular, warning system, cycling

Poglavje 1

Uvod

Umetna inteligenca (UI) je zelo široko interdisciplinarno področje računalništva, ki proučuje teoretični in praktični razvoj sistemov, ki se vedejo, kot da bi razpolagali z inteligenco. Povezuje se z mnogimi drugimi vedami, kot so matematika, jezikoslovje, psihologija, nevrologija, strojništvo, ekonomija in statistika.

Sistemi UI so večinoma razviti kot sestavni deli zapletenejših sistemov, ki jim dodajo »inteligenco«, npr. jim omogočijo razumevanje naravnega jezika, sklepanje s pridobljenim znanjem ali učenje in prilagoditev.

Dandanes lahko s pomočjo umetne inteligence avtomobil brez človeškega posredovanja prevozi puščavo, upošteva prednosti v nesemaforiziranih križiščih in se nemoteno giblje tudi v urbanih območjih. Maja 2012 so v zvezni državi Nevada (ZDA) izdali prvo registrsko tablico avtonomnemu vozilu [27], kar dokazuje zrelost implemeniranih postopkov umetne inteligence v teh vozilih, med drugim postopkov računalniškega vida.

Tehnološki napredek pri računalniškem vidu se trenutno uporablja le pri avtomobilih. Za izboljšanje varnosti na cestah bi lahko razvili aplikacijo tudi za kolesarje, ki pri udeležbi v prometu nimajo nobenega opozorilnega sistema, večina nima niti vzvratnih ogledal.

Razlog, zakaj takšnih aplikacij še ni, tiči v procesorski zahtevnosti postopkov, ki se uporabljajo. Z rastjo procesorske zmogljivosti v manjših napravah,

npr. »pametnih« telefonih, ki jih lahko uporabljamo na kolesih, pa je uporaba teh postopkov mogoča. Razvili smo aplikacijo, ki se namesti v pametni telefon in kolesarje opozarja na nevarnosti v prometu.

Ključni del sistema za zaznavanje okolice pri nekaterih avtomobilih brez voznika ter nekaterih zračnih, zemeljskih in podmorskih vozilih brez posadke je knjižnica Open Source Computer Vision Library (OpenCV), ki predstavlja osnovo, na kateri bomo gradili tudi naš postopek.

Naša rešitev uporablja nekaj osnovnih metod zgoraj omenjene knjižnice OpenCV. Problem rešujemo postopno. Najprej skušamo prepoznati cestišče in ugotoviti, kje je ponorna točka. Te podatke nato uporabimo za prepoznavanje oddaljenosti nevarnosti in zaznavanje prehitevanja.

Rezultati postopka so vzpodbudni, vendar nam, kot velikokrat pri podobnih postopkih pri avtomobilih, največ težav pri zaznavi in s tem največ šuma povzročajo sence na cestišču.

V naslednjem poglavju bomo navedli dosedanje raziskave na tem področju in sorodne aplikacije (poglavje 2). Natančneje bomo definirali problem, ki smo ga reševali, in opisali specifične težave pri kolesarjih (poglavje 3). Sledil bo podrobnejši opis knjižnice OpenCV in uporabljenih metod (poglavje 4).

Opisali bomo razvoj postopka za rešitev problema (poglavje 5), njegovo teoretično podlago in praktično implementacijo rešitve v aplikaciji eyeCycle (poglavje 6). Ocenili bomo uspešnost postopka na testnih posnetkih (poglavje 7) in predlagali mogoče izboljšave (poglavje 8).

Poglavje 2

Pregled področja

Registracija prvega avtonomnega vozila kaže na zrelost uporabljenih postopkov in tehnologij pri njegovi izdelavi. V nadaljevanju bomo pregledali zgodovino avtonomnih vozil, njihovo povezavo z našim problemom in aplikacije na tem področju.

Začetki razvoja avtonomnih vozil segajo v 80. leta prejšnjega stoletja. Zanimanje za avtonomna vozila se do danes ni zmanjšalo, saj imajo takšna vozila veliko prednosti. Strnemo jih lahko v naslednje točke:

- večja izkoriščenost cestnega omrežja,
- manjša poraba goriva in energije,
- zmanjšano število potrebnega osebja za mobilnost ljudi in blaga,
- izboljšana kakovost mobilnosti in
- izboljšanje varnosti v primerjavi s sedanjo ureditvijo.

Največji projekt razvoja, povezan z avtonomnimi avtomobili, ki ga je financirala Evropska komisija, je bil projekt EUREKA Prometheus[1]. Pri izvedbi so sodelovali številne univerze in proizvajalci avtomobilov, rezultat projekta pa je bila opredelitev najsodobnejše (vrhunske) tehnologije na področju avtonomnih vozil [6][2].

[1] PROgraMme for a European Traffic of Highest Efficiency and Unprecedented Safety

Omenili bomo dve avtonomni vozili, ki izhajata iz projekta EUREKA Prometheus, VaMP in ARGO.

VaMP je bil eno izmed prvih avtonomnih vozil. Gre za model avtomobila mercedes 500 SEL, ki ga je razvil Ernst Dickmanns v sodelovanju s podjetjem Mercedes-Benz [10]. Avtomobil je omogočal krmiljenje, pospeševanje in zaviranje s pomočjo računalniških ukazov. Prevozil je tisoč kilometrov na pariški večpasovni avtocesti v srednje gostem prometu s hitrostmi do 130 km/h. Prikazal je avtonomno vožnjo po prostih voznih pasovih, vožnjo v koloni, samodejno sledenje drugim vozilom ter menjavo pasov na levo in desno s samodejnim prehitevanjem. VaMP ni uporabljal GPS-a kot novejša vozila, temveč se je zanašal le na računalniški vid.

Avtonomni avtomobil ARGO [31] je nastal kot nadaljevanje raziskovanja projekta Prometheus, ki se je končal s prej omenjeno vožnjo v Parizu. Avtomobil so razvili na Katedri za informacijsko tehnologijo (*Dipartimento di Ingegneria dell'Informazione*) na Univerzi v Parmi. Robustnost uporabljenih postopkov pri razvoju je, poleg vožnje po avtocestah in hitrih cestah, omogočila tudi testiranje vozila ARGO na zunajmestnih podeželskih cestah. Leta 1998 so demonstrirali delovanje vozila ARGO, ki je z avtonomno vožnjo prevozilo dva tisoč kilometrov po vsej Italiji.

Leta 2004 je ameriška agencija Defense Advanced Research Projects Agency (DARPA) financirala prvo tekmovanje na dolge razdalje za avtonomne avtomobile na svetu, imenovano DARPA Grand Challenge[2], in zmagovalcu obljubila milijon ameriških dolarjev. Tekmovanja nobeno vozilo ni uspešno končalo, zato so naslednje leto nagrado povišali na dva milijona ameriških dolarjev. Za zmago je bilo treba prevoziti 240 km po puščavskih poteh. Progo je uspešno prevozilo pet vozil, med njimi pa je bilo najhitrejše in zato zmagovalno vozilo ekipe univerze Standford, ki jo je vodil Sebastian Thrun [16].

DARPA Grand Challenge je bil izziv umetne inteligence in ne izključno računalniškega vida. Poglavitni del izziva je bil s pomočjo senzorjev pridobiti čim več podatkov, odstraniti šum in sprejemati najboljše odločitve glede na

[2]http://www.darpagrandchallenge.com/

razmere, to je voziti avtomobil po »puščavskih« poteh.

Naslednji izziv agencije DARPA je potekal leta 2007 pod imenom DARPA Urban Challenge in je od vozil zahteval avtonomno delovanje v modelu urbanega okolja. Proga je bila dolga 96 km. Pravila so vključevala obvezno upoštevanje vseh prometnih predpisov, hkrati pa zahtevala še pravilno upoštevanje prednosti v križiščih, vključevanje v promet in upoštevanje drugih udeležencev v prometu. Zadnje je še posebej težavno, saj morajo vozila sprejemati »inteligentne« odločitve v realnem času glede na dejanja drugih vozil. Na prejšnjih tekmovanjih (in drugih preteklih projektih) je bila namreč interakcija med vozili in drugimi udeleženci v prometu minimalna. Progo je uspešno prevozilo šest vozil. Zmagovalec je bila ekipa Tartan Racing [18][17]. Vozilo univerze Standford pod vodstvom Sebastiana Thruna je bilo drugo najhitrejše vozilo [13].

Sebastian Thrun je vodja projekta tudi pri Googlovem avtomobilu brez voznika (angl. *Google driverless car*), ki je prvo registrirano vozilo brez voznika. Del ekipe med drugim predstavljajo inženirji, ki so sodelovali v tekmovanjih agencije DARPA. To kaže, da je bilo znanje iz tekmovanj DARPA preneseno v prakso in da je tekmovanje pripomoglo tudi k napredku tehnologije. Slika 2.1 prikazuje prvi registrirani avtomobil brez voznika. Vozilo ima vgrajene naslednje senzorje: kamere, senzor LIDAR (angl. *Light Detection And Ranging*) na strehi, radarske senzorje na sprednjem delu vozila in položajni senzor na enem izmed zadnjih koles, ki pomaga pri določitvi položaja vozila na zemljevidu [27].

Kot smo nakazali v uvodu, smo se osredotočili na kolesarje in njihove težave v prometu, zato ne bomo imeli na razpolago toliko senzorjev in kamer kot vozila, ki smo jih omenili do zdaj. Podmnožico postopkov, ki so bili uporabljeni v avtomobilih, bomo poskušali vključiti v aplikacijo na pametnem telefonu (zakaj pametni telefon, bomo opisali v naslednjem poglavju).

Obstoječe aplikacije za kolesarje na tem področju uporabljajo le globalni pozicijski sistem (GPS) za določanje položaja in hitrosti, prevožene razdalje ter za računanje statistike o trenutni poti. Nobena aplikacija ne opozarja

Slika 2.1: Prvi registrirani avtomobil brez voznika. S puščicama sta označena senzor LIDAR (angl. *Light Detection And Ranging*) na strehi in položajni senzor na levem zadnjem kolesu.

Slika 2.2: Zaslonski posnetek aplikacije iOnRoad. Na levi strani zgoraj je navedena trenutna hitrost vožnje. Zeleno pobarvano cestišče označuje vozni pas in varnostno razdaljo, nad njo pa je prikazan čas, ki je potreben za prevoz razdalje pri trenutni hitrosti.

kolesarja na nevarnosti, temveč ga kvečjemu usmerja po zemljevidu.

Za avtomobile obstaja omembe vredna tržna aplikacija za mobilne telefone iOnRoad [26], ki zaznava menjavo pasu, opozarja na varnostno razdaljo in izpisuje zavorni čas glede na hitrost potovanja (Slika 2.2). Aplikacija, tako kot naša, le opozarja na nevarnosti. Za zaznavanje uporablja optimizirano knjižnico FastCV [25], ki del algoritmov obdeluje strojno (vendar le na določeni strojni opremi, za zdaj pa tudi le na platformi Android), zato so optimizacije občutne. Poleg tega pa je telefon med uporabo priklopljen na napajanje v avtomobilu (kot pri nekaterih drugih poskusih [14]) in se ne napaja iz baterije.

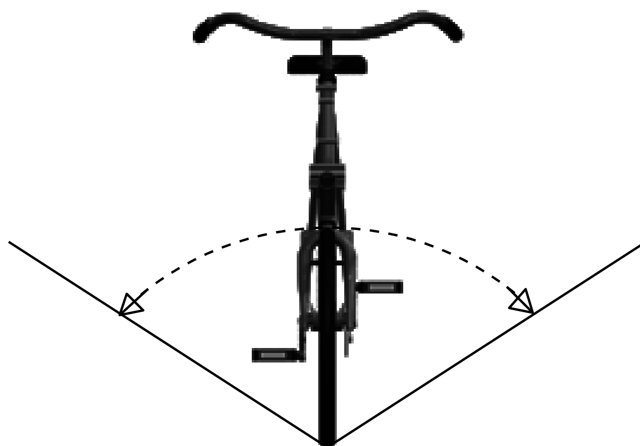
Na podlagi pregleda področja smo se odločili, da bomo za razvoj naše aplikacije uporabili knjižnico Open Source Computer Vision Library (OpenCV), ki jo je uporabil tudi zmagovalec tekmovanja DARPA Grand Challenge, ekipa univerze Stanford [8].

Poglavje 3

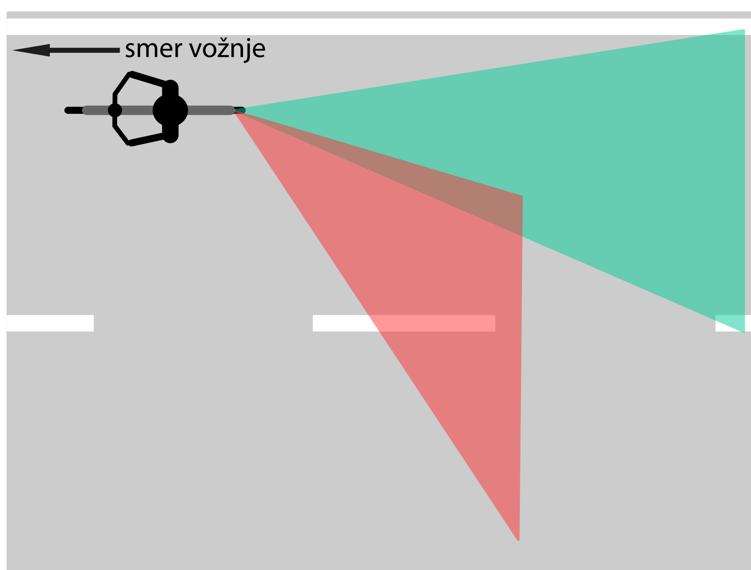
Opis problema

Namen diplomskega dela je bil raziskati in razviti aplikacijo za povečanje varnosti kolesarjev v prometu. V primerjavi z avtomobili se pri kolesih srečamo z dodatnimi, specifičnimi težavami. Slika 3.1 prikazuje zibanje kolesa, ki ga pri avtomobilih ne srečamo. Ti imajo namreč štiri kolesa, kar jim omogoča večjo stabilnost. Kolo je zaradi slabšega oziroma neobstoječega vzmetenja tudi bolj občutljivo za tresljaje. Krajša medosna razdalja kolesu omogoča krajši obračalni krog, kar povzroča hitrejšo spremembo slike, ki jo posname kamera, pritrjena na kolesu. Slika 3.2 prikazuje dve območji, ki predstavljata nevarnost, če se v njih pojavi vozeče vozilo. Približevanje vozil je označeno z zeleno barvo. Druga nevarnost, nevarnost prehitevajočega vozila, je na sliki označena z rdečo barvo. Vse to bomo morali upoštevati pri razvoju našega postopka. Za razliko od avtomobilov pa imajo kolesa še dodatne omejitve. V avtomobil lahko namestimo enega ali več zmogljivih računalnikov. Ti s pomočjo različnih senzorjev (dve ali več kamer, laser, radar, GPS itd.) omogočajo natančnejše meritve in obdelavo podatkov s procesorsko zahtevnejšimi postopki.

Na kolo pa ne moremo namestiti vseh teh naprav in senzorjev. Lahko bi sicer poskusili, vendar bi bilo kolo na koncu verjetno neuporabno. Poleg tega bi morale biti takšne naprave in senzorji tudi snemljivi, da jih ne bi kdo poskušal ukrasti. Idealno rešitev vidimo v uporabi pametnih telefonov,



Slika 3.1: Zibanje kolesa



Slika 3.2: Območja nevarnosti pri kolesih

saj jih zaradi njihove majhnosti zlahka namestimo na kolo. Poleg tega se njihova procesorska zmogljivost redno povečuje, tako da so dandanes že sposobni obdelati tudi nekatere izmed zapletenejših algoritmov računalniškega vida. Novejši modeli imajo vgrajene tudi dodatne senzorje, kot so npr. pospeškometer ali žiroskop, ki bi jih lahko uporabili za natančnejšo obdelavo podatkov.

S temi lastnostmi pametnih telefonov lahko poskusimo narediti sistem, ki bo kolesarje opozarjal na prej opisane nevarnosti.

Ker imamo na napravi na voljo le eno kamero in ker na nevarnosti le opozarjamo (s kolesom ne bomo upravljali), so merila za natančnost nižja. Poleg tega že vnaprej pričakujemo, da bo ena izmed ključnih ovir pri uporabni vrednosti aplikacije poraba baterije. Del razvoja bo torej tudi iskanje sprejemljivega razmerja med uspešnostjo zaznave in porabo baterije, ki je posredno povezana s procesorsko zahtevnostjo postopka. Potencial za izboljšave vidimo v OpenCL (angl. *Open Computing Language*), ki bo v prihodnosti omogočal, da bomo del procesorskega dela CPE lahko dodelili vse bolj zmogljivim grafičnim procesorjem. Kolesarja bomo na nevarnosti opozarjali z zvočnimi piski. Večja ko bo nevarnost, krajši bo časovni presledek med posameznimi piski.

Poleg naprav samih pa je zanimiva insfrastruktura, ki se gradi okoli njih. Trgovine z aplikacijami omogočajo majhnim razvijalcem dosegljivost svetovnega trga. Teoretično bi lahko našo aplikacijo uporabljalo več tisoč uporabnikov. Če bi v sistem vgradili možnost pošiljanja povratnih informacij, bi ga lahko še dodatno izpopolnili, saj bi s tem izrazito povečali učno množico, oziroma povečali količino podatkov, ki so nam na voljo, kot če podatke pridobivamo le na eni napravi v laboratoriju.

Zaradi zmogljivosti naprav, možnosti distribucije in osebnega izziva smo se odločili za platformo iOS, aplikacijo pa bomo objavili v trgovini Apple App Store.

Poglavje 4

Uporabljene metode OpenCV in njihova teoretična podlaga

Open Source Computer Vision Library (OpenCV) je odprtokodna knjižnica programskih metod, ki se osredotoča predvsem na obdelavo slike v realnem času. Razvil jo je Intel in je prosta za uporabo pod odprtokodno licenco BSD [29]. Projekt je bil sprva pobuda Intel Research Intiative za izboljšanje in optimizacijo procesorsko intenzivnih aplikacij. K projektu so največ prispevali številni strokovnjaki za optimizacijo iz Intela Rusija in skupina Intel's Performance Library Team. Sprva so bili cilji projekta opisani na naslednji način[5]:

- napredovanje raziskav, povezanih z računalniškim vidom, z zagotavljanjem ne samo odprte, temveč tudi optimizirane kode za osnovne gradnike računalniškega vida;
- širjenje znanja o računalniškem vidu z zagotavljanjem skupne infrastrukture, na kateri bi lahko razvijalci gradili naprej, s tem pa ustvarili lažje berljivo in prenosljivo kodo;
- napredovanje komercialnih aplikacij računalniškega vida s pomočjo prenosljive, zmogljive kode, ki bo na voljo brezplačno, z nezavezujočo licenco o odprtosti oziroma brezplačnosti.

Od svoje prve alfa različice, izdane leta 1999, je bil OpenCV uporabljen v veliko aplikacijah, kot so npr. zmanjševanje šuma v bolniških slikah, varnostni sistemi, sistemi za odkrivanje vdorov, sistemi za nadzor kakovosti v proizvodnji, kalibriranje kamer, vojaške aplikacije ter v zračnih, zemeljskih, in podvodnih vozilih brez posadke. OpenCV je bil tudi ključni del sistema vida robota Stanleyja, razvitega na univerzi Standford, ki je zmagal na tekmovanju DARPA Grand Challenge [30].

Knjižnica OpenCV je sestavljena iz velikega števila metod. Mi smo uporabili le majhno podmnožico teh metod. V nadaljevanju bomo opisali metode v knjižnici OpenCV, ki smo jih uporabili za gradnjo našega postopka, in teoretično opisali ozadje njihovega delovanja.

4.1 threshold

```
double threshold (InputArray src, OutputArray dst,
                  double thresh, double maxval, int type);
```

Pragovna operacija *threshold* se uporablja za nastavljanje fiksnega praga na enokanalni sliki. Običajno jo uporabljamo zato, da iz sivinske (angl. *grayscale*) slike dobimo binarno sliko, ali za odstranitev šuma, tj. odstranitev točk, ki imajo premajhno vrednost. Vrednost vsake točke se izračuna s spodnjo enačbo.

$$dst(x,y) = \begin{cases} maxval & \text{če je } src(x,y) > thresh \\ 0 & \text{sicer} \end{cases}$$

4.2 floodFill

```
int floodFill (InputOutputArray image,
               Point seedPoint, Scalar newVal, out Rect* rect=0,
               Scalar loDiff=Scalar(), Scalar upDiff=Scalar(),
               int flags=4);
```

Metoda *floodFill()* prebarva dele slike, ki so med seboj povezani. Prebarvanje začne v sejalni točki (angl. *seed point*) z navedeno barvo (*newVal*). Medsebojna povezanost je definirana glede na skladnost barve sosednjih slikovnih točk. Točka (x, y) pripada prebarvanemu predelu, če izpolnjuje spodnje pogoje, kjer vsaka neenačba ustreza enemu izmed treh barvnih kanalov RGB (rdeča, zelena, modra).

$$src(seed.x, seed.y)_r - loDiff_r \leq src(x, y)_r \leq src(seed.x, seed.y)_r + upDiff_r$$

$$src(seed.x, seed.y)_z - loDiff_z \leq src(x, y)_z \leq src(seed.x, seed.y)_z + upDiff_z$$

$$src(seed.x, seed.y)_m - loDiff_m \leq src(x, y)_m \leq src(seed.x, seed.y)_m + upDiff_m$$

To pomeni, da se točka doda prebarvanemu predelu, če je njena barva dovolj podobna barvi sejalne točke.

4.3 add

```
void add(InputArray src1, InputArray src2, OutputArray dst,
         InputArray mask=noArray(), int dtype=−1);
```

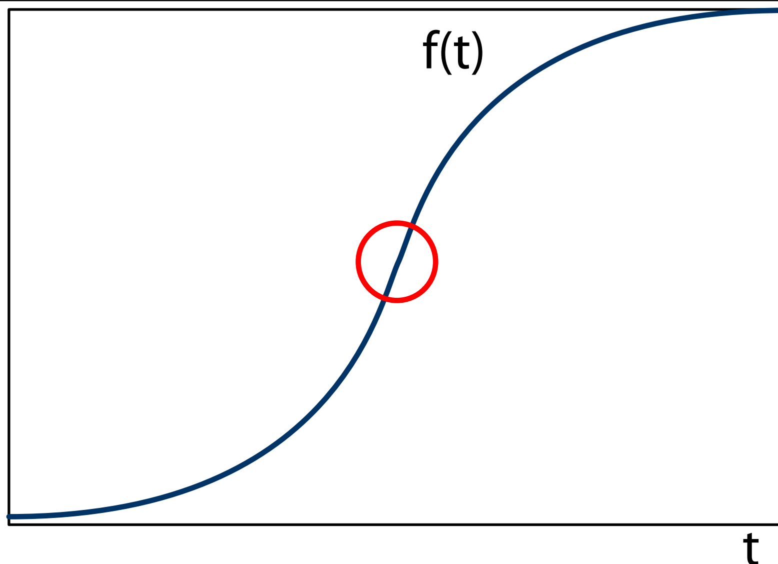
Metoda *add()* izračuna matrično vsoto $src1$ in $src2$ ter jo zapiše v dst.

$$dst = src1 + src2$$

4.4 erode

```
void erode (InputArray src, OutputArray dst, InputArray kernel,
            Point anchor=Point(−1,−1), int iterations=1,
            int borderType=BORDER_CONSTANT,
            const Scalar& borderValue=morphologyDefaultBorderValue());
```

Metodo *erode()* smo uporabili za odstranitev šuma. Njeno delovanje je ekvivalento izračunu lokalnega minimuma v območju jedra (privzeto velikosti 3 x 3). Operacija *erode()* ustvari novo sliko iz izvirnika po naslednjem algoritmu:



Slika 4.1: Sprememba svetlosti v sliki

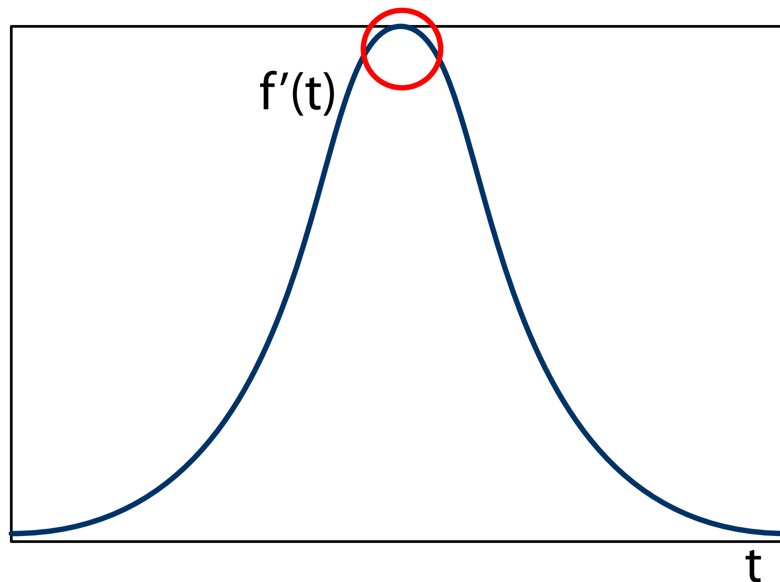
Medtem ko jedro pomikamo čez sliko, izračunamo minimalno vrednost točk, ki jih jedro prekriva, in zamenjamo vrednost pod točko sidranja (*anchor*) s to minimalno vrednostjo.

4.5 Sobel

```
void Sobel (InputArray src, OutputArray dst, int ddepth,
                    int dx, int dy, int ksize=3,
                    double scale=1, double delta=0,
                    int borderType=BORDER_DEFAULT);
```

Če želimo zaznati robove, moramo najprej definirati, kaj je rob. Na robu se zgodi nenadna velika sprememba v svetlosti slike. Slika 4.1 prikazuje to spremembo svetlosti.

»Skok« v svetlosti lahko bolje vidimo z uporabo prvega odvoda, kjer se sprememba pokaže kot lokalni ekstrem (v našem primeru maksimum), kar je prikazano na Sliki 4.2 .



Slika 4.2: Z odvodom prikazana sprememba svetlosti v sliki

Iz zgornje razlage sledi, da je dobra metoda za iskanje robov v sliki iskanje točk, ki imajo gradient večji od sosednjih točk oziroma večji od nekega praga. Sobelov operator uporabi ravno to dejstvo in robove išče tako, da izračuna približek gradienta v vsaki točki v sliki. Približek gradienta računa tako, da sliki (I) doda dve konvolucijski maski (v smeri x in y):

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I$$

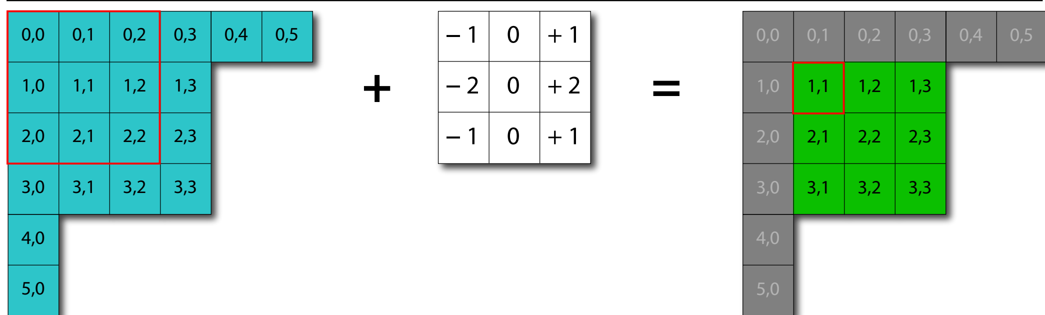
Približek velikosti gradienta izračuna z enačbo:

$$G = \sqrt{G_x^2 + G_y^2}$$

Mi smo za izračun uporabili poenostavljeno enačbo:

$$G = |G_x| + |G_y|$$

Praktična implementacija Sobelovega operatorja je zelo preprosta. Ustvarimo dva filtra in z njima obdelamo vsako točko v naši sliki, začenši z leve



Slika 4.3: Praktična implementacija Sobelovega operatorja

proti desni. Slika 4.3 prikazuje takšno obdelavo slike. Pozoren bralec lahko opazi, da s takšnim postopkom ne moremo oceniti prve in zanje vrstice ter prvega in zadnjega stolpca, ker je filter matrika velikosti 3 x 3. Izhodna slika bi bila torej manjše velikosti kot vhodna. OpenCV že vključuje rešitev za to težavo, vendar je tukaj ne bomo omenjali, saj ni bistvena za razumevanje delovanja Sobelovega operatorja.

4.6 Canny

```
void Canny (InputArray image, OutputArray edges,
                    double threshold1, double threshold2,
                    int  apertureSize=3, bool L2gradient=false);
```

Cannyjev detektor robov, med drugim poznan tudi kot *optimalni detektor*, je razvil John F. Canny leta 1986. Detektor skuša zadovoljiti tri glavna merila:

- nizko stopnjo napake: odkrivanje le dejanskih (obstoječih) robov;
- dobro lokalizacijo: razdalja med zaznanimi robovi in dejanskimi robovi mora biti minimalna;
- minimalno odzivnost: detektor se na vsak rob odzove le enkrat.

Postopek je naslednji:

1. *Canny()* najprej filtrira šum. To stori z normalnim (Gaussovim) filtrom, npr. z jedrom velikosti 5.

$$K = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

2. Nato poišče svetlostni gradient slike. Za to uporablja postopek, podoben Sobelovemu. Sliki dodamo dve konvolucijski maski (v smeri x in y).

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

   Poiščemo jakost in smer gradienta z enačbama:

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

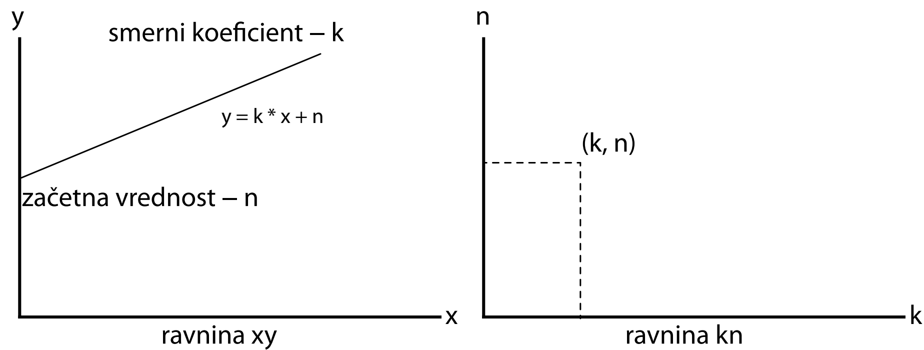
   Smer zaokrožimo na enega izmed štirih kotov (0, 45, 90, 135).

3. Odpravimo vse točke, ki niso maksimum. S tem odstranimo točke, za katere ne štejemo, da so del roba. Ostanejo nam le tanke črte (kandidati za robove).

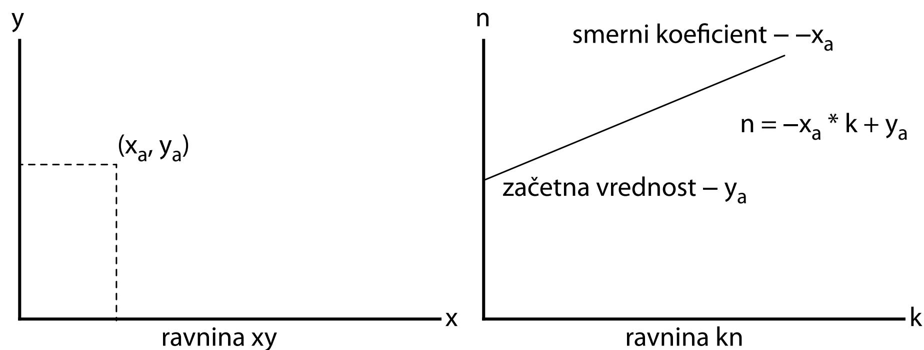
4. Zadnji korak je upragovljenje s histerezo. *Canny*() uporabi dva pragova (zgornjega in spodnjega):

   - če je gradient točke višji kot zgornji prag, jo sprejmemo kot rob;
   - če je gradient točke nižji kot spodnji prag, jo zavrnemo;
   - če je gradient točke med pragoma, jo bomo sprejeli kot rob le, če je stična/povezana s točko, ki je nad zgornjim pragom.

Canny je priporočil razmerje med zgornjim in spodnjim pragom med $2:1$ in $3:1$.



Slika 4.4: Preslikava premice iz ravnine xy v ravnino kn

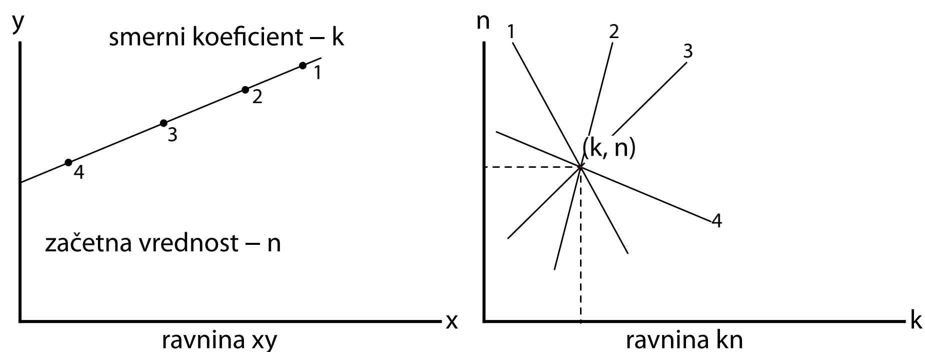


Slika 4.5: Preslikava točke iz ravnine xy v ravnino kn

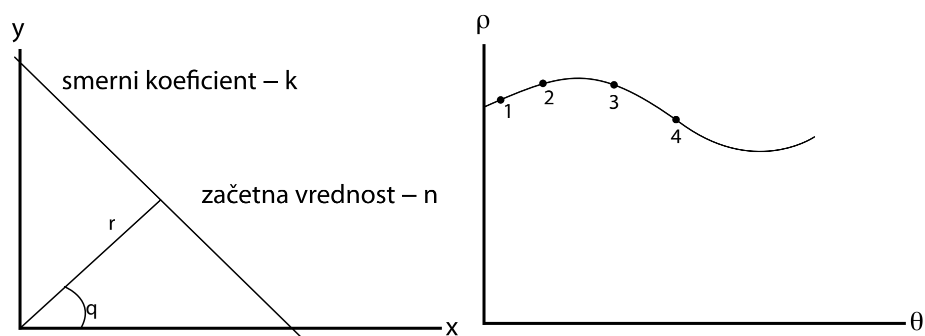
4.7 Hough

```
void HoughLinesP (InputArray image, OutputArray lines,
                  double rho, double theta, int threshold,
                  double minLineLength=0, double maxLineGap=0);
```

Osnovna predpostavka Houghove transformacije je, da je vsaka točka v binarni sliki del množice morebitnih črt. Če vsako črto parametriziramo z njeno začetno vrednostjo n in smernim koeficientom k, potem se točka v izvirni sliki (ravnina xy) preslika v množico točk (premico) v ravnini kn. Točka v ravnini kn pa ustreza vsem točkam, ki ležijo na isti premici v ravnini xy. Slika 4.4 prikazuje preslikavo premice iz ravnine xy v točko v ravnini kn.



Slika 4.6: Preslikava več točk iz ravnine xy v ravnino kn. Presek premic (lokalni maksimum) je točka (k, n), ki definira premico v ravnini xy.



Slika 4.7: Levo – grafični prikaz premice v polarnih koordinatah. Desno – točke, preslikane v ravnino $p\theta$.

Slika 4.5 pa prikazuje preslikavo točke iz ravnine xy v premico v ravnini kn.

Če torej pretvorimo vsako neničelno točko v vhodni sliki v množico točk (premico) v izhodni sliki in seštejemo prekrivanja, potem se bodo črte, ki se pojavijo v vhodni (ravnina xy) sliki, prikazale kot lokalni maksimumi v izhodni (ravnina kn) sliki (Slika 4.6). Ker seštevamo prekrivanja vsake točke, ravnino kn navadno imenujemo tudi zbirna ravnina.

Parametrizacija v ravnino kn je lahko v računalniških sistemih težavna, saj se smerni koeficient navpične črte nagiba proti neskončnosti, zato sta njeno shranjevanje in obdelava otežena ali celo nemogoča. Zaradi tega se parametrizacija točke pri numeričnem računanju nekoliko razlikuje. Pri računanju zato raje uporabljamo polarne koordinate. Enačba premice je v tem primeru

$$p = x \cos \theta + y \sin \theta,$$

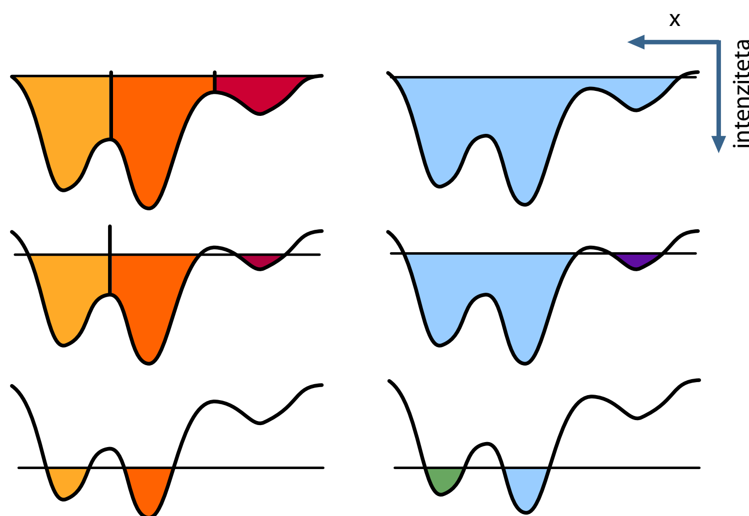
kjer je kot θ polarni kot, p pa radij. Enačbo grafično prikazuje Slika 4.7.

Z uporabo te enačbe pride do spremembe pri preslikavi točke v premico iz ravnine xy v (polarno) ravnino $p\theta$. Premica v ravnini xy še vedno ustreza točki v (polarni) ravnini $p\theta$. Vendar točka v ravnini xy zdaj ustreza sinusni krivulji v (polarni) ravnini $p\theta$, kot prikazuje Slika 4.7.

4.8 watershed

```
void watershed (InputArray image, InputOutputArray markers);
```

V praktičnih primerih računalniškega vida želimo segmentirati sliko. V našem primeru želimo sliko segmentirati na dva dela: na cesto in okolico. Ena izmed tehnik, ki se za to uporablja, je algoritem watershed (razvodje). Algoritem najprej uporabi gradient svetlosti v sliki in z njim generira »relief« svetlosti. Posledica tega je, da se na območjih, kjer ni teksture oziroma je regija enakomerna, oblikujejo doline ali bazeni, prevladujoče črte v sliki pa se oblikujejo v »hribe«. Algoritem nato zaporedoma poplavi povodja, kot je prikazano na Sliki 4.8 (slika je zrcaljena čez os x, zaradi boljšega prikaza



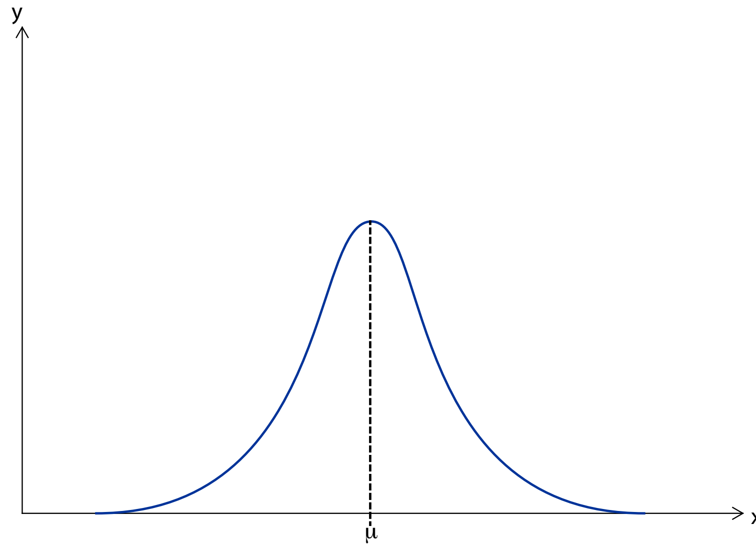
Slika 4.8: Na levi prikaz polnjenja območji z razvodji (pregradami), na desni prikaz, kako bi se območja polnila brez pregrad.

»poplave«). Začne na od uporabnika določenih (ali z algoritmom specificiranih) točkah in nadaljuje, vse dokler se dve različni regiji ne srečata. Medtem ko se slika »polni«, točke srečanja posebej označi. Za regije, ki se združijo na tako pridobljenih oznakah, štejemo, da spadajo skupaj. Na ta način bazeni, povezani z orientacijsko oznako, postanejo združeni s to oznako. Nato sliko segmentiramo glede na ustrezno označena območja.

4.9 KalmanFilter

```
KalmanFilter (int dynamParams, int measureParams, int controlParams=0, int type
        =CV_32F);
const Mat& predict (const Mat& control=Mat());
const Mat& correct (const Mat& measurement);
```

Kalmanov filter smo uporabili za sledenje ključnim točkam in odstranitev šuma pri tem sledenju. Je algoritem za rekurzivno obdelavo/procesiranje podatkov. Generira optimalno oceno stanja sistema glede na sklop meritev. Uporablja sklop meritev, ki vsebujejo šum (naključne spremembe) in so opa-



Slika 4.9: Normalna (Gaussova) porazdelitev – $\mathcal{N}(\mu, \sigma^2)$

zovane v daljšem časovnem obdobju. Poda nam natančnejšo oceno stanja sistema kot ocena, ki temelji le na eni meritvi. Ker je rekurziven, ne potrebuje hrambe vseh prejšnjih meritev in obdelave le-teh v vsakem koraku [20].

Osnovna predpostavka je, da je sistem linearen ter so vse napake in meritve definirane z normalno (Gaussovo) porazdelitvijo (Slika 4.9). Zaradi poenostavitve bomo za prikaz delovanja uporabili enodimenzionalen Kalmanov filter (v našem postopku smo uporabili večdimenzionalnega, kar se pogosteje uporablja v praksi). Gaussova porazdelitev je definirana kot

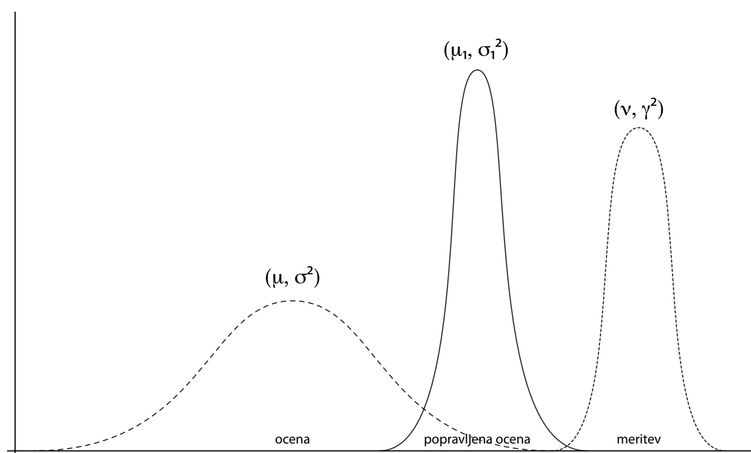
$$\mathcal{N}(\mu, \sigma^2),$$

kjer je μ pričakovana vrednost, σ^2 pa varianca.

Če definiramo našo dosedanjo oceno sistema kot $\mathcal{N}(\mu, \sigma^2)$ in našo izmerjeno vrednost kot $\mathcal{N}(\nu, \gamma^2)$, lahko novi vrednosti za μ_1 in σ_1^2 izračunamo po naslednjih enačbah:

$$\mu_1 = \frac{\mu\gamma^2 + \nu\sigma^2}{\sigma^2 + \gamma^2}$$

$$\sigma_1^2 = \frac{1}{\frac{1}{\sigma^2} + \frac{1}{\gamma^2}}$$



Slika 4.10: Prikaz koraka korekcije (angl. *measurement update* ali *correction*)

Slika 4.10 prikazuje oceno in izmerjeno vrednost ter njuno združitev. Kot lahko vidimo, je varianca ocene (σ^2) večja kot varianca meritve (γ^2), saj meritvi občutno bolj zaupamo. Zanimivo je, da je varianca združenih količin (σ_1^2) manjša kot katera izmed drugih dveh varianc (σ^2, γ^2). Ta del postopka imenujemo korekcija (angl. *correction* ali *measurement update*).

Drugi del postopka se imenuje napoved (angl. *prediction* ali *motion update*). V tem delu napovemo, kakšna bo naslednja meritev. Za to uporabimo spodnji dve enačbi.

$$\mu_1 = \mu + \nu$$

$$\sigma_1^2 = \sigma^2 + \gamma^2$$

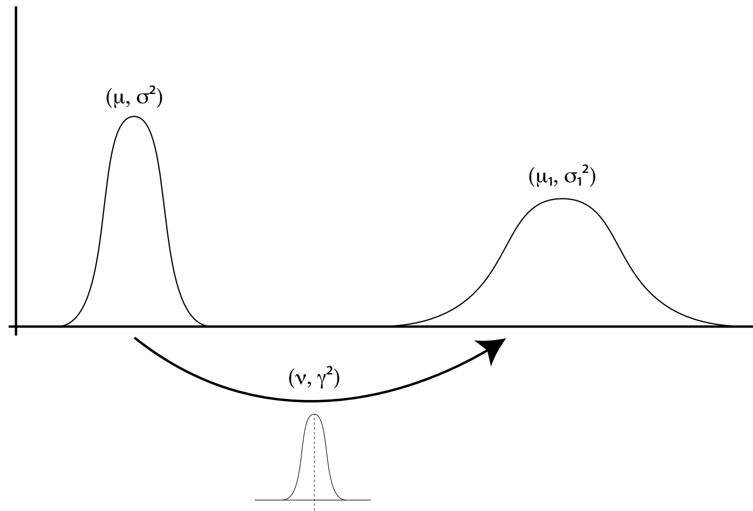
Korak napovedi je prikazan na Sliki 4.11.

Ta dva postopka nato ponavljamo.

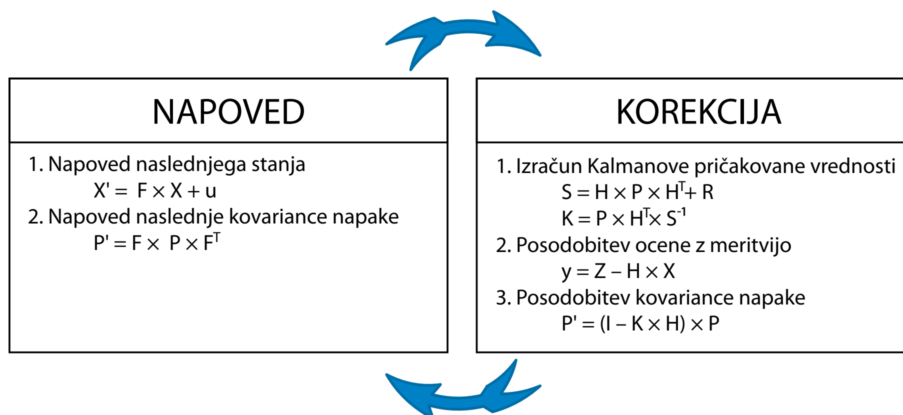
Večdimenzionalni Kalmanov filter ima še to lastnost, da lahko iz izmerjenih količin in stanja sistema izračuna tudi neizmerjene količine (npr. izračuna hitrost, izmeri pa le položaj) [11].

Slika 4.12 prikazuje cikel večdimenzionalnega Kalmanovga filtra, kjer so oznake vrednosti naslednje:

X – ocena



Slika 4.11: Prikaz koraka napovedi (angl. *motion update* ali *prediction*)



Slika 4.12: Prikaz cikla večdimenzionalnega Kalmanovega filtra

P – kovarianca napake

F – matrika prehoda stanj

u – vektor gibanja

Z – meritev

H – funkcija meritve

R – šum meritve

I – identiteta

4.10 goodFeatureToTrack

```
void goodFeaturesToTrack (InputArray image, OutputArray corners,
                          int  maxCorners, double qualityLevel, double minDistance,
                          InputArray mask=noArray(), int blockSize=3,
                          bool useHarrisDetector=false, double k=0.04);
```

Če želimo slediti točkam, moramo najprej definirati, katere lastnosti ima točka, ki ji je smiselno slediti. V knjigi *Learning OpenCV* [5] avtor definira: ”*... točka, ki jo izberemo, mora biti enolična ali skoraj enolična, in mora biti parametrizirana na takšen način, da jo lahko primerjamo z drugimi točkami v neki drugi sliki.*”

Bralec si lahko več o izbiri točk za sledenje ter o težavah in rešitvah, povezanih s tem problemom, prebere v *Learning OpenCV* [5] (316, 317).

Točke, katerim sledimo, izberemo s pomočjo metode *goodFeaturesToTrack*. Metoda *goodFeatureToTrack* privzeto uporablja Shi-Tomasijevo metodo, ki je malce izboljšana Harrisova metoda za izbiro točk.

Naš cilj je torej poiskati dele slike (ali »okna«), ki generirajo veliko spremembo, ko se premaknejo.

Harrisov detektor oglišč je le matematičen način za določitev oken, ki povzročijo veliko spremembo, ko se premaknejo v katerokoli smer. Vsakemu oknu se določi ocena R, na podlagi katere lahko ugotovimo, ali gre za oglišče ali ne. To lahko zapišemo z enačbo:

$$E(u, v) = \sum_{x,y} w(x, y)[I(x + u, y + v) - I(x, y)]^2,$$

kjer oznake pomenijo:

E – razlika med izvornim in premaknjenim oknom

u – sprememba okna v smeri x

v – sprememba okna v smeri y

w(x, y) – okno na položaju (x, y)

I(x, y) – svetlost izvorne slike

I(x + u, y + v) – svetlost premaknjenega okna

S pomočjo Taylorjeve vrste in nekaj matematičnih izpeljav dobimo naslednjo enačbo:

$$E(u,v) \approx [u,v]\left(\sum w(x,y)\begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}\right)\begin{bmatrix} u \\ v \end{bmatrix} \tag{4.1}$$

Matriko, ki vsebuje vsoto, poimenujemo M in jo zapišemo kot

$$M = \sum w(x,y)\begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$

Torej lahko enačbo 4.1 zapišemo kot

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

Pri ugotavljanju primernosti okna si pomagamo z lastnima vrednostma matrike (λ_1, λ_2). Rezultat R izračunamo za vsako okno po naslednji enačbi:

$$R = \det M - k(\text{trace} M)^2$$

$$\det M = \lambda_1\lambda_2$$

$$\text{trace} M = \lambda_1 + \lambda_2,$$

kjer je k empirična konstanta ($0,04$–$0,06$ ali tudi do $0,15$).

Razlika med Harrisovo metodo in Shi-Tomasijevo metodo, ki jo privzeto uporablja OpenCV, je v tem, da Shi-Tomasi za izračun R primerja najmanjšo med lastnima vrednostma in preveri, ali je večja od nekega minimuma. V članku sta Shi in Tomasi dokazala, da ta sprememba za izračun R deluje veliko bolje kot Harrisova metoda [15].

$$R = min(\lambda_1, \lambda_2)$$

4.11 cornerSubPix

```
void cornerSubPix (InputArray image, InputOutputArray corners,
                   Size winSize, Size zeroZone,
                   TermCriteria criteria);
```

Metoda *goodFeaturesToTrack* vrne točke s celoštevilskimi koordinatami. Ker želimo natančnejše meritve, uporabimo še metodo *cornerSubPix*. Ključna zamisel metode *cornerSubPix* je pridobitev več enačb in reševanje sistema enačb. Z uporabo prej izračunanega približka točke oglišča in več točk v njegovi bližini tako pridobimo več enačb. Vse te enačbe pa so enake nič. Za vsako sosednjo točko imamo namreč dve možnosti:

- če sosednja točka (q) leži na istem robu kot opazovana točka (p), potem je vektor gradienta pravokoten na razliko njunih vektorjev in je skalarni produkt enak 0;
- če sosednja točka (q) ne leži na robu, je gradient enak 0 in je zatorej skalarni produkt enak 0.

Skalarni produkt je v obeh primerih enak 0:

$$< \nabla I(p), q - p >= 0$$

Dobljena rešitev po zgornjem postopku je natančna na več decimalnih mest, kar poveča natančnost v primerjavi s celoštevilsko rešitvijo, ki smo jo dobili z uporabo metode *goodFeaturesToTrack* [5].

4.12 calcOpticalFlowPyrLK

```
void calcOpticalFlowPyrLK (InputArray prevImg, InputArray nextImg,
                           InputArray prevPts, out InputOutputArray nextPts,
                           OutputArray status, OutputArray err,
                           Size winSize=Size(21,21), int maxLevel=3,
                           TermCriteria criteria=TermCriteria(TermCriteria::COUNT
                                  +TermCriteria::EPS, 30, 0.01),
                           int flags=0, double minEigThreshold=1e-4);
```

Za sledenje uporabimo OpenCV-jevo metodo *calcOpticalFlowPyrLK*, ki uporablja piramidni algoritem Lucas-Kanade (LK) (najprej uporabi večje obsege slike, nato obdeluje vse manjše)[5]. Ta je izboljšava nepiramidnega LK-algoritma. LK-algoritem temelji na predpostavki konstantne svetlosti slikovne točke, kar lahko prikažemo s spodnjo enačbo[9].

$$I(x, y, t) = I(x + dx, y + dy, t + dt) \tag{4.2}$$

Če predpostavimo manjše premike, lahko uporabimo Taylorjevo vrsto, s katero dobimo naslednji rezultat:

$$I(x+dx, y+dy, t+dt) = I(x, y, t)+\frac{\partial I}{\partial x}(x, y, t)dx+\frac{\partial I}{\partial y}(x, y, t)dy+\frac{\partial I}{\partial t}(x, y, t)dt \tag{4.3}$$

Z združitvijo obeh enačb (4.2 in 4.3) dobimo enačbo za optični tok:

$$\frac{\partial I}{\partial x}(x, y, t)dx + \frac{\partial I}{\partial y}(x, y, t)dy + \frac{\partial I}{\partial t}(x, y, t)dt = 0, \tag{4.4}$$

ki jo lahko zapišemo tudi kot:

$$\frac{\partial I}{\partial x}(x, y, t)v_x + \frac{\partial I}{\partial y}(x, y, t)v_y + \frac{\partial I}{\partial t}(x, y, t) = 0, \tag{4.5}$$

kjer je $v = [v_x, v_y]^T$ vektor hitrosti ob položaju in času (x, y, t).

Enačbo 4.5 lahko zapišemo tudi v diferencialni obliki:

$$[I_x, I_y]\begin{bmatrix} v_x \\ v_y \end{bmatrix} + I_t = 0, \tag{4.6}$$

kjer so I_x, I_y in I_t parcialni odvodi $I(x, y, t)$. To je sistem enačb z dvema neznankama v_x, v_y, ki ni rešljiv. Problem je poznan tudi kot problem zaslonke algoritmov optičnega toka. Za rešitev problema potrebujemo dodatne enačbe, ki jih vpeljemo z dodatnimi omejitvami. Za reševanje problema zaslonke algoritem LK predpostavlja prostorsko povezanost. Če je vektor hitrosti $v = [v_x, v_y]^T$ v majhnem oknu z velikostjo n točk okoli točke (x, y)

konstanten, dobimo:

$$\begin{bmatrix} I_{x1} & I_{y_1} \\ I_{x2} & I_{y_2} \\ \vdots & \vdots \\ I_{x_n} & I_{y_n} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} + \begin{bmatrix} I_{t_1} \\ I_{t_2} \\ \vdots \\ I_{t_n} \end{bmatrix} = 0$$

Ta sistem enačb lahko zapišemo v matrični obliki:

$$Av + b = 0$$

Za rešitev tega sistema uporabimo metodo najmanjših kvadratov:

$$v = (A^T A)^{-1} A^T (-b)$$

Za izpeljavo enačb in delovanje algoritma so potrebne tri predpostavke.

1. Konstantna svetlost: vsaka slikovna točka dotičnega predmeta v prizoru ne spreminja videza, ko se (morda) premakne iz okvira v okvir. V našem primeru to za sivinsko sliko pomeni, da domnevamo, da se svetlost slikovne točke, medtem ko ji sledimo iz okvira v okvir, ne spreminja.

2. Časovna obstojnost ali »majhni premiki«: premikanje dela slike (površine) se v časovnem obdobju dogaja počasi. V praksi to pomeni, da so časovni koraki, glede na obseg gibanja v sliki, dovolj hitri, da se objekt ne premakne veliko iz okvira v okvir.

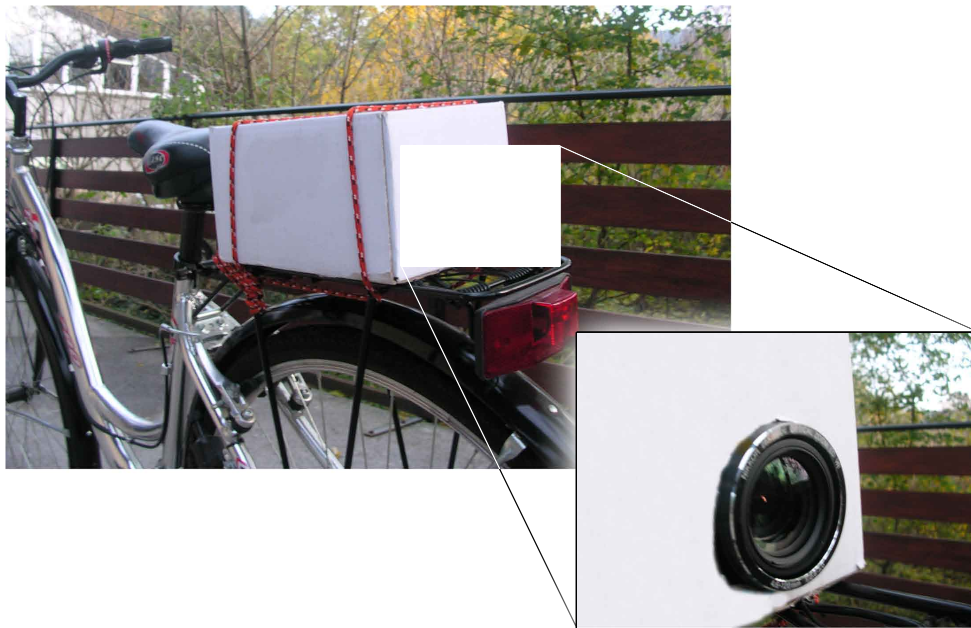
3. Prostorska povezanost: sosednji točki v prizoru pripadata isti površini, se podobno gibljeta in se preslikata v bližnje točke na slikovni ravnini.

Poglavje 5

Razvoj postopka

Razvoj smo začeli na namiznem računalniku. Delovno okolje smo virtualizirali. S tem smo dosegli počasnejše delovanje ter zmanjšali razliko med hitrostjo namiznega in mobilnega procesorja. Najprej smo morali pridobiti testne posnetke. Videokamero smo pritrdili na kolo, kot prikazuje Slika 5.1, in snemali dogajanje za kolesom med vožnjo. Začetni razvoj je potekal v programskem jeziku C# (avtorju v času razvoja bolj domač, v primerjavi z Objective-C, ki je programski jezik na ciljni napravi). Uporabili smo Microsoftovo razvojno okolje Visual C# 2008 Express Edition. Ker pa OpenCV direktno ne podpira jezika C#, smo uporabili Emgu CV [24]. To je ovojnica .NET za knjižnico OpenCV, ki omogoča klicanje metod OpenCV iz .NET združljivih jezikov, kot je npr. C#.

Zavedajoč se dejstva, da je mobilni procesor na končni napravi počasnejši od razvojnega (kar bo delalo počasi oziroma sprejemljivo hitro na razvojnem procesorju, ne bo delalo oz. bo delalo nesprejemljivo počasi na mobilnem procesorju), smo začeli iskati najbolj optimalno rešitev našega problema. Pred tem smo problem razdelili na manjše, bolj obvladljive podprobleme. Najprej smo morali na posnetku prepoznati območje, na katerega bomo pozorni, tj. območje zanimanja ali angleško ROI (Region of interest). Privzeli smo, da je naše območje zanimanja kar cesta (cestišče). Torej smo reševali problem segmentacije ceste in določitve njenih robov (Poglavje 5.1). Naslednji

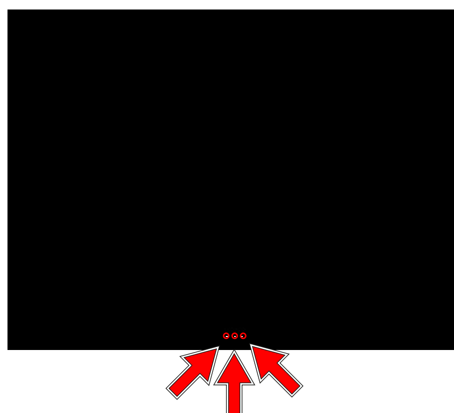


Slika 5.1: Videokamera, pritrjena na kolo za pridobivanje posnetkov

podproblem je bilo zaznavanje približevanja vozil (Poglavje 5.2), za njim pa zaznavanje prehitevanja (Poglavje 5.3). Vse tri podprobleme in naše rešitve zanje bomo opisali v nadaljevanju.

5.1 Segmentacija ceste in določitev njenih robov

Ker bo telefon lociran na zadnjem delu kolesa, lahko predpostavimo, da se bo cestišče vedno začelo na robu naše pridobljene slike in izginilo nekje na sredini slike. Ta predpostavka je bila uporabljena že v mnogih obstoječih projektih [3] [4]. Določili bomo točko na sredini slike, malo nad spodnjim robom, ki nam bo služila za izhodišče pri določevanju cestišča. Pri tem lahko nastane težava zaradi specifične lastnosti kolesa, in sicer da kolo ne vozi po sredini vozišča, temveč po robu oziroma celo po črti na robu vozišča. Za rešitev tega problema bomo poleg točke na sredini uporabili še dve točki, ki



Slika 5.2: Tri točke, označene s puščicami, za katere predpostavljamo, da so ob vsakem času na cestišču.

sta malce oddaljeni od sredine na levi in desni [1].

Slika 5.2 prikazuje zgoraj omenjene točke.

Za boljšo definicijo območja cestišča izbrane tri točke razširimo z metodo *floodFill* (Slika 5.3), ki nam območje cestišča pobarva glede na razlike med barvami točk. Dobljeno območje skupaj z predpostavko, da cestišče izgine na polovici slike, uporabimo kot vhod v metodo *watershed*, ki nam sliko razdeli na dve območji, in sicer na cesto in okolico. Slika 5.4 prikazuje, kako nam *watershed* segmentira cesto. V nadaljevanju bomo opisali, kako s pomočjo segmentirane ceste določimo robova vozišča [7].

Podobno barvno razvrščanje, kot je algoritem *watershed*, so uspešno uporabili zmagovalci DARPA Challengea. Pri njih so bili vhodni podatki pridobljeni iz drugih senzorjev [16].

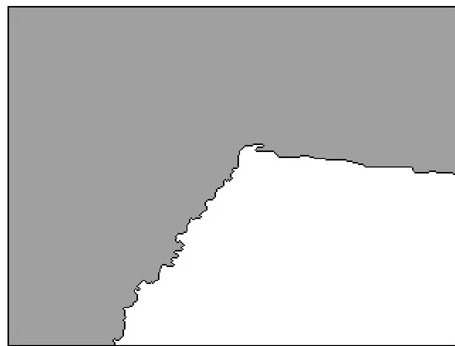
V našem primeru imamo na razpolago le eno kamero, zato namesto natančnih meritev uporabimo prej omenjene predpostavke (območje, pridobljeno z metodo *floodFill*, in črto, ki omejuje okolico, prikazano na Sliki 5.5).

Slika 5.6 prikazuje mogočo uporabo metode *watershed* za poenostavljeno določitev cestišča, ki bi lahko bila uporabljena v DARPA Challengeu [5].

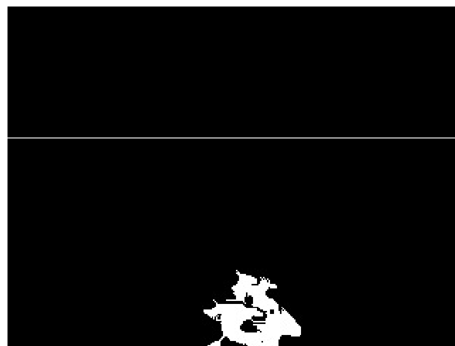
S pomočjo metode *Canny* iz segmentirane slike pridobimo mejo med območjema ceste in okolice, kar prikazuje Slika 5.7. Na tej sliki poženemo

(a) (b)

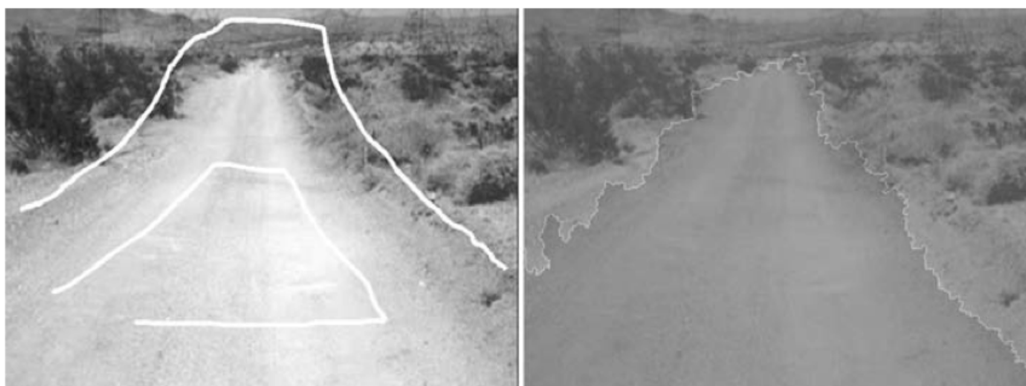
Slika 5.3: (a) izbrane točke, razširjene z metodo *floodFill*; (b) območje, razširjeno z metodo *floodFill*



Slika 5.4: Segmentacija ceste in okolice z metodo *watershed*

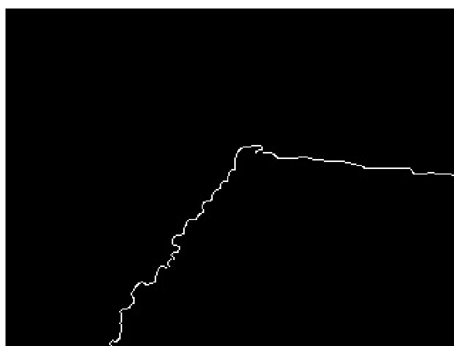


Slika 5.5: Vhodna slika (markerji) za metodo *watershed*



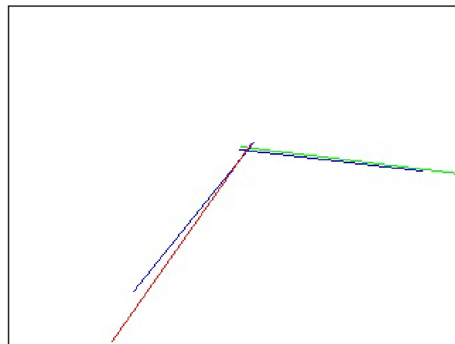
(a) (b)

Slika 5.6: Mogoča uporaba metode *watershed* za segmentacijo ceste: slika z markerji (a) in segmentirana cesta (b)



Slika 5.7: Meja med cesto in okolico, izrisana s pomočjo metode *Canny*

Slika 5.8: Segmentirano cestišče in s pomočjo metode *HoughLinesP* zaznane črte



Slika 5.9: Z rdečo označen levi rob ceste, z zeleno desni in z modro drugi kandidati za robove ceste

še metodo *HoughLinesP*, ki nam iz slike izloči črte, ki se pojavijo v njej. Črte so kandidati za robove vozišča. S pomočjo vnaprej določenih meril, ki obsegajo dolžino črte, njen naklon in položaj ter njeno začetno in končno točko, izberemo oziroma določimo levi in desni rob cestišča. Nato izračunamo presečišče levega in desnega roba cestišča, presečišče desnega roba cestišča z desnim ali spodnjim robom slike ter presečišče levega roba cestišča z levim oziroma spodnjim robom slike. Tem presečiščem nato sledimo. Postopek sledenja je podrobneje opisan v nadaljevanju.

Slika 5.8 prikazuje segmentirano cestišče in okolico ter zaznane črte. V modri barvi so narisane zaznane črte, z rdečo kandidat za levi rob, z zeleno

pa kandidat za desni rob. Za boljšo preglednost Slika 5.9 prikazuje enako obarvane črte brez segmentiranega cestišča in okolice.

Za dobro zaznane robove (oziroma dobro meritev v postopku sledenja, opisanem v nadaljevanju) štejemo le primer, ko hkrati zaznamo levi in desni rob ter je njuno presečišče na območju slike [12].

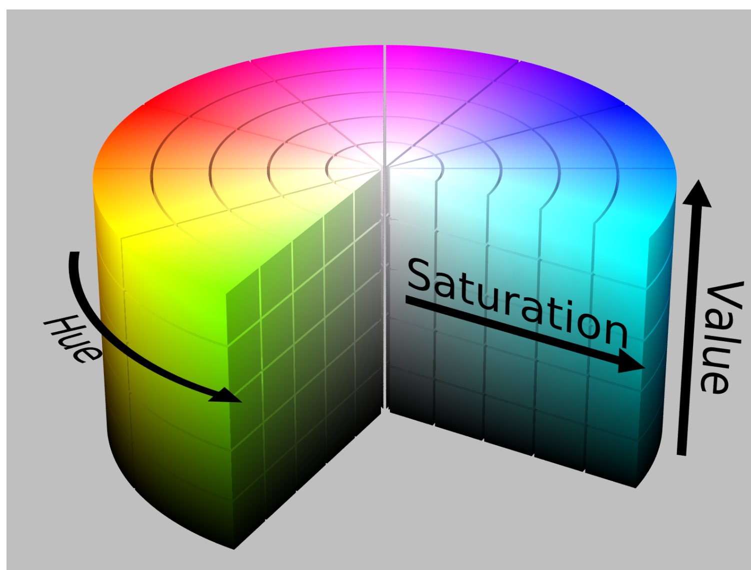
Pri definiranju robov vozišča se soočamo z veliko šuma. Ker moramo slediti mejnim točkam (levi, desni in presečišču), jim sledimo s Kalmanovim filtrom.

Kalmanov filter uporabljamo tudi zato, da je slika bolj stabilna in smo natančnejši pri meritvi ceste oziroma presečišča njenih robov, saj na tej točki slonita druga dva dela postopka. Preden nadaljujemo s preostalima postopkoma, moramo pridobiti več meritev. Ker potrebujemo več dobrih meritev, od zagona (začetka postopka) poteče nekaj časa, preden je aplikacija pripravljena za uporabo.

5.2 Zaznavanje približevanja

V prejšnjem poglavju smo s pomočjo izbranih postopkov določili tri točke: levi rob, desni rob in presečišče. Presečišče nam predstavlja točko, od katere se nam bodo objekti približevali, zato je pomembno, da območje med presečiščem in spodnjim robom slike nima ovir. Da lahko to opazujemo, moramo vhodno sliko binarizirati, tj. spremeniti v sliko, sestavljeno le iz belih in črnih točk, kjer ohranimo ovire oz. nevarnosti. Postopek se razlikuje za barvni in sivinski vhod. Oba postopka bomo opisali v nadaljevanju.

1. Barvni vhod segmentiramo s pomočjo slike, pridobljene iz barvnega prostora HSV (*hue*, *saturation*, *value*). Barvni prostor HSV se bistveno razlikuje od obče poznanega barvnega prostora RGB (*red*, *green*, *blue*). V tem je slika razdeljena na tri kanale, vsakega za eno barvo: rdečo, zeleno in modro. V prostoru HSV pa je informacija o intenziteti (svetlost) ločena od barvne informacije (barvitost). Tridimenzionalni prikaz barvnega prostora HSV je valj s centralno navpično osjo, ki predstavlja



Slika 5.10: Barvni prostor HSV; *saturation* je nasičenost, *value* je svetlost, *hue* je odtenek.

svetlost (Slika 5.10). Odtenek je opredeljen kot kot v območju $[0,2\pi]$ glede na rdečo, kjer ima rdeča barva kot 0, zelena $\frac{2\pi}{3}$, modra $\frac{4\pi}{3}$ in rdeča spet 2π. Nasičenost je globina ali čistost barve in se meri v oddaljenosti od osrednje osi z vrednostmi od 0 v centru do 1 na zunanji površini [19].

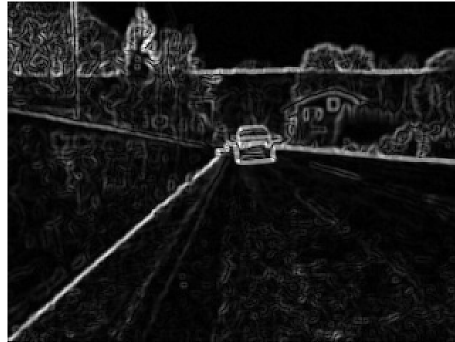
Originalno sliko v formatu RGB (Slika 5.11) naprej pretvorimo v prostor HSV. Uporabimo kanal nasičenosti (ang. *saturation*), prikazan na Sliki 5.12, ki ga še dodatno obdelamo z metodo *threshold*. Rezultat je prikazan na Sliki 5.13.

2. Sivinski vhod je za razliko od barvnega sestavljen le iz enega kanala, zato za njegovo obdelavo uporabimo drugačen postopek. Sivinsko sliko najprej obdelamo s Sobelovim operatorjem, s katerim zaznavamo robove. Nad izhodno sliko iz Sobelovega operatorja uporabimo še operacijo *erode()*. Slika 5.14 prikazuje Sobelov gradient vhodne slike, Slika 5.15 pa izhodno sliko po operaciji *erode()*.

Slika 5.11: Originalna barvna slika iz kamere

Slika 5.12: S-kanal barvnega prostora HSV originalne Slike 5.11

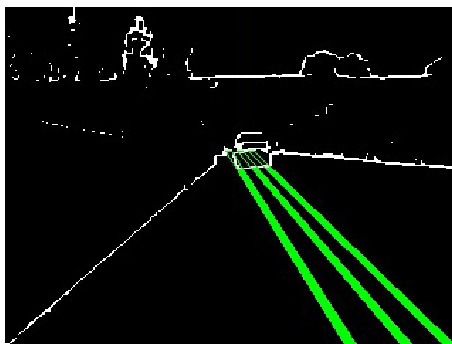
Slika 5.13: Slika S-kanala iz Slike 5.12 po uporabi pragovne metode *threshold*



Slika 5.14: Originalna Slika 5.11 po uporabi Sobelovega operatorja

Slika 5.15: Slika 5.14 po uporabi metode *erode*

Slika 5.16: Slika 5.15 z dodanimi mejnimi črtami pred barvanje z metodo *floodFill*



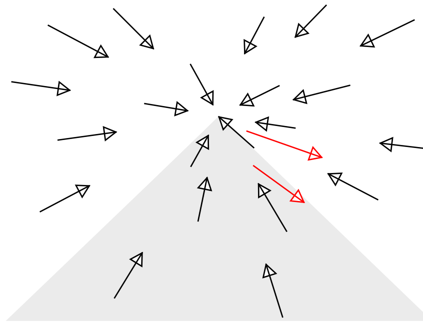
Slika 5.17: Mejne črte po uporabi metode *floodFill*

Kot že omenjeno, iz prejšnjih dveh postopkov dobimo ustrezno črno-belo binarno sliko. Bližino izmerimo s pomočjo metode *floodFill*. *FloodFill* smo izbrali, ker omogoča dvoje: s pomočjo parametrov *upDiff* in *loDiff* lahko natančno definiramo, kako bo barvanje potekalo, metoda pa nam poleg izhodne slike vrne tudi mere in koordinate pobarvanega območja (parameter *rect*), iz katerih lahko nato izračunamo razdaljo do ovire. S tem načinom zaznamo le ovire, ki presegajo širino med dvema robnima črtama. Zaradi tega robne črte ne smejo biti preveč narazen. Da pokrijemo celotno območje za kolesom, ta postopek ponovimo trikrat, kot to prikazujeta Slika 5.16 in Slika 5.17. Med tremi meritvami upoštevamo najmanjšo meritev.

Izmerjeno razdaljo do ovire primerjamo z razdaljo do presečišča robov ceste. Razmerje med njima uporabimo za opozarjanje na nevarnosti. Različne stopnje nevarnosti so natančneje opisane v poglavju 6.3 in navedene v Tabeli 6.1.

5.3 Zaznavanje prehitevanja

Za zaznavanje prehitevanja uporabljamo optični tok. Metode za zaznavo optičnega toka lahko razdelimo v dve skupini, na gosti in redki optični tok. Poglavitna razlika med njima je, da gosti tok izračuna vektor hitrosti za vsako točko in je procesorsko zahtevnejši, zato bomo uporabili redkega, ki



Slika 5.18: Vektorji premikanja. Črni prikazujejo premikanje v smeri ponorne točke, rdeči pa premikanje v nasprotni smeri.

sledi le točkam, ki jih predhodno izberemo. Če pravilno izberemo točke, je postopek robustnejši in zanesljiv.

Preprosteje povedano, sledimo določenim točkami, ki smo jih izbrali, in ugotavljamo, v katero smer so se premaknile.

Z metodama *goodFeatureToTrack* in *cornerSubPix* smo definirali, katere točke je smiselno izbrati. Zdaj jim moramo še slediti. S pomočjo metode *calcOpticalFlowPyrLK* izračunamo vektorje premika za vsako točko, ki ji sledimo. Ponorno točko uporabimo kot koordinatno izhodišče in sliko razdelimo na štiri kvadrante. Vektorji, ki predstavljajo prehitevanja, so tisti, ki se v drugem kvadrantu (desno spodaj) gibljejo v smeri, nasprotni ponorni točki, kot je prikazano na Sliki 5.18.

Pri računanju optičnega toka imamo opravka z veliko količino šuma, ki ga skušamo upoštevati oz. zanemariti. To storimo s pomočjo vnaprej določenih meril (npr. večkratno pojavljanje iste nevarnosti, dolžina vektorja, število vektorjev na bližnjem območju), s katerimi filtriramo vektorje premikanja in občutno zmanjšamo šum.

Prvi dve predpostavki, konstantno svetlost in časovno obstojnost, ki sta navedeni na koncu poglavja 4.12, moramo upoštevati tudi pri gradnji našega postopka. Prvo predpostavko upoštevamo tako, da programsko izklopimo avtomatsko prilagajanje ekspozicije (osvetlitve). Metode *calcOpticalFlowPyrLK*,

zaradi njene počasnosti (zahtevnosti), ne poganjamo na vsaki sličici. Da izpolnimo pogoje druge predpostavke, pa moramo zato vedno hraniti prejšnjo sliko (tj. sliko, ki jo bomo skupaj s trenutno poslali kot parameter algoritmu – parameter *prevImg*).

Poglavje 6

Implementacija postopka: aplikacija eyeCycle

Ker smo se odločili, da bomo končno aplikacijo razvili za platformo iOS, smo za implementacijo in razvoj le-te uporabili razvojno okolje Xcode. Ker pričakujemo, da bo aplikacija procesorsko požrešna, določimo, da bo aplikacija uporabna le na napravah z novejšo verzijo iOS-a, saj predpostavljamo, da so novejše naprave večinoma hitrejše. Novejše različice okolja Xcode omogočajo uporabo ARC (angl. *Automatic Reference Counting*), ki smo ga uporabili tudi mi. ARC poenostavi upravljanje pomnilnika, saj nam ni treba poskrbeti za njegovo ročno dealokacijo.

Razvojno okolje Xcode vsebuje tudi simulator ciljne naprave, imenovan iOS Simulator. Težava tega programa je, da je le vizualni simulator. Izvajanje naše kode se namreč ne simulira (poganjanje kode za procesor ARM), ampak je koda prevedena za arhitekturo x86 in se poganja na procesorju v našem namiznem računalniku. To je prispevalo tudi k neprijetnemu presenečenju (počasno delovanje), ko smo aplikacijo preizkusili na mobilni napravi in mobilnem procesorju.

Ključni del našega postopka je, kot smo večkrat omenili, knjižnica Open Source Computer Vision Library. Poiskati smo morali način, kako knjižnico uporabiti tudi v okolju Xcode. Poskusili smo več načinov, opisanih na sple-

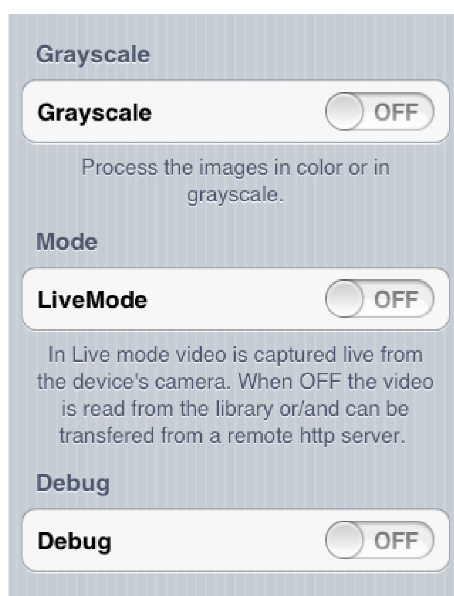
Slika 6.1: Ikona aplikacije eyeCycle

tni strani o računalniškem vidu [22], ki je priporočala povezovanje na zglavne datoteke, vendar ob prenosu na napravo rešitev ni delovala. Končno rešitev smo dobili na spletni strani [23], ki opisuje, kako ustvariti programsko ogrodje OpenCV, domorodno za iOS oziroma Xcode. To ogrodje oziroma postopek za njegovo kreiranje pa je zdaj na voljo tudi na uradni strani knjižnice OpenCV[29].

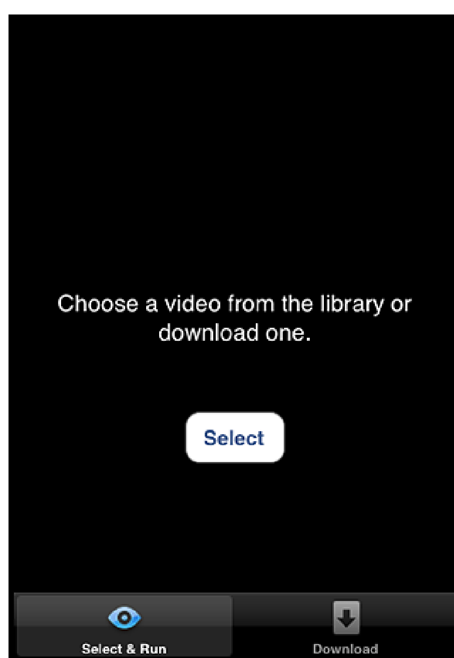
Postopek zaznave cestišča in njegovih robov (poglavje 5.1), zaznavanje prebliževanja (poglavje 5.2) in zaznavanje prehitevanja (poglavje 5.3) smo implementirali v aplikaciji, imenovani eyeCycle. Aplikacija omogoča dva načina delovanja: način obdelovanja posnetkov in t. i. način v živo. V načinu obdelovanja posnetkov obdelujemo posnetke, shranjene v knjižnici naprave, način v živo pa obdeluje, kot samo ime pove, sliko v živo iz videokamere na napravi, kjer se poganja.

Slika 6.1 prikazuje ikono, ki smo jo naredili za aplikacijo.

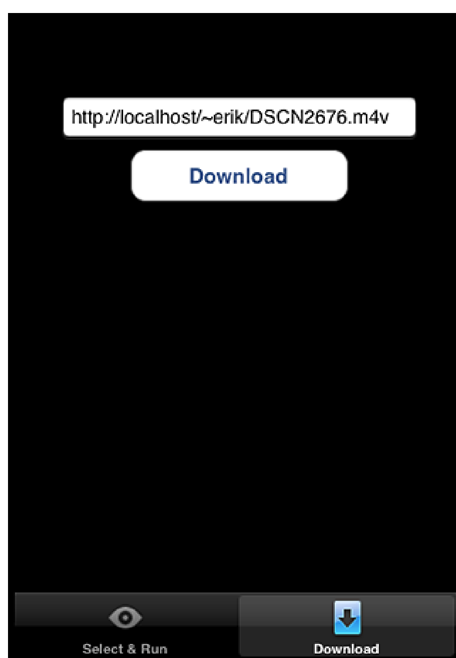
Med dvema načinoma uporabe uporabnik preklaplja s stikalom v nastavitvah naprave, ki ga prikazuje Slika 6.2. Oba načina bomo podrobno opisali v nadaljevanju. Poleg teh dveh načinov uporabnik lahko spreminja tudi način, kako kamera zajema vhod, sivinsko ali barvno, in nastavitev, ali se aplikacija izvaja v načinu za razhroščevanje (poglavje 6.3).



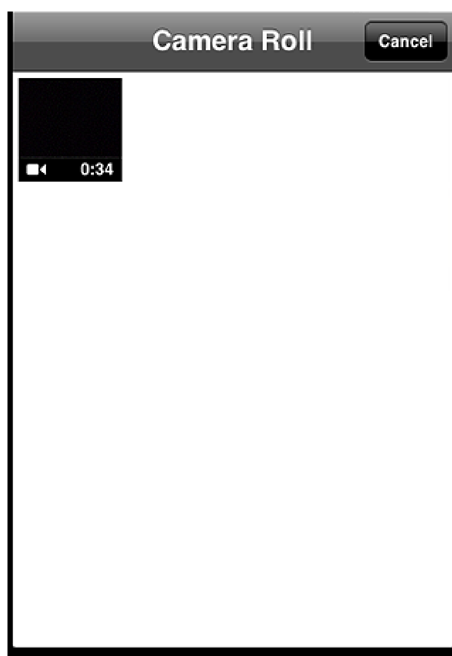
Slika 6.2: Meni nastavitev aplikacije



Slika 6.3: Začetni meni v načinu obdelave posnetkov



Slika 6.4: Možnost prenosa posnetkov na napravo z oddaljenega strežnika s pomočjo protokola http



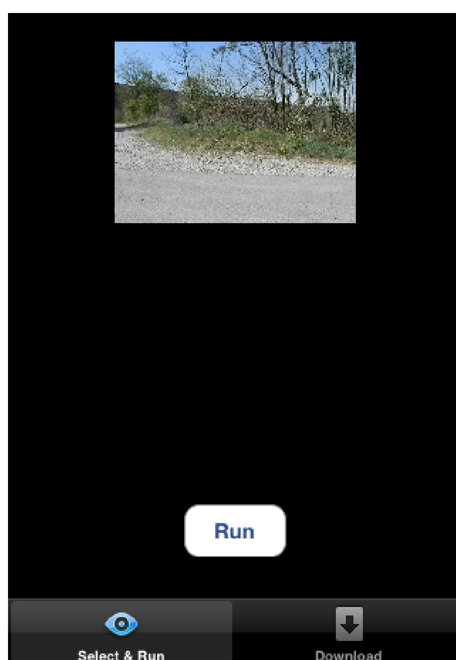
Slika 6.5: Knjižnica posnetkov na napravi

6.1 Obdelava posnetkov

V načinu obdelovanja posnetkov aplikacija omogoča izbiro posnetka iz knjižnice (Slika 6.3). Če posnetka še nimamo v knjižnici, aplikacija omogoča prenos posnetka s pomočjo protokola http. Posnetek mora biti v formatu *MPEG4 Video* (M4V). V polje za naslov vnesemo celoten URL-naslov posnetka in pritisnemo na gumb »Download« (Slika 6.4). Aplikacija javi morebitno napako v prenosu oz. nedostopnost strežnika. V nasprotnem primeru nas obvesti o uspešnem prenosu posnetka. Posnetek po končanem prenosu tudi avtomatsko naloži in pripravi za obdelavo.

Če imamo posnetek že v knjižnici, samo pritisnemo gumb »Select« (Slika 6.3) in aplikacija nam prikaže posnetke iz naše knjižnice (Slika 6.5). Za vsak posnetek imamo na voljo predogled. Ko ga izberemo, ga aplikacija pripravi za obdelavo.

Ko je posnetek pripravljen za obdelavo, imamo na voljo gumb »Run«,



Slika 6.6: Prikaz naloženega posnetka pred začetkom obdelave s prvo sličico v predogledu in gumbom *Run* za zagon obdelave

nad katerim je prikazana prva sličica v posnetku (Slika 6.6). Po pritisku na gumb »Run« poženemo postopek obdelave posnetka.

Posnetki, shranjeni v knjižnici, imajo vsaj 25 sličic na sekundo (angl. *frames per second* ali fps). Delovanje postopka bi bilo zelo počasno, če bi obdelovali vsako sličico, zato postopek pohitrimo in obdelujemo le vsako deseto sličico. S tem popravkom dosežemo hitrost posnetka, ki je primerljiva z resničnostjo. V nasprotnem primeru bi bilo delovanje toliko upočasnjeno, da aplikacija ne bi bila uporabna. Obdelovanje posameznih sličic v posnetku nato poteka enako kot v načinu v živo, kar bomo podrobneje opisali v nadaljevanju.

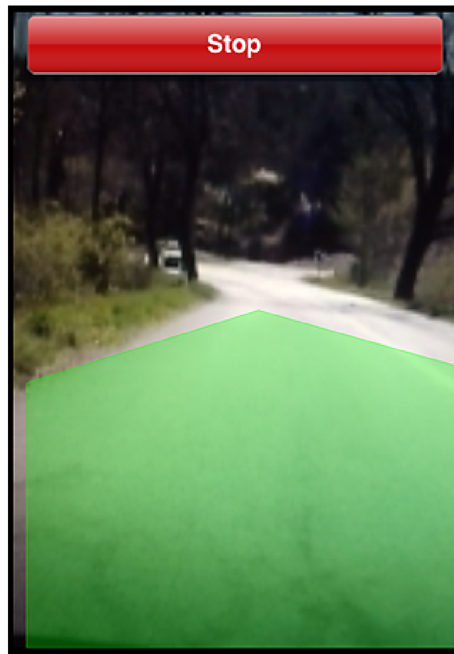
6.2 LiveMode

V načinu v živo uporabnika najprej z opozorilom opozorimo, da je to testna aplikacija in da lahko pod določenimi pogoji deluje nezanesljivo.

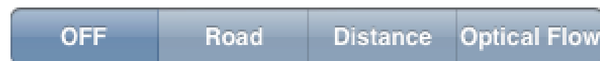
Uporabnik ima nato na voljo zelen gumb z napisom »Start«, ki požene postopek obdelovanja žive slike. Gumb »Start« se spremeni v rdeč gumb »Stop«. Ob pritisku nanj se postopek ustavi. Ko je postopek zaustavljen, je na napravi omogočeno samodejno nastavljanje žarišča (ang. *autofocus*) in samodejno nastavljanje ekspozicije (ang. *autoexposure*), če naprava to podpira. Po zagonu postopka pa *focus* in *exposure* fiksiramo. To moramo narediti za pravilno delovanje algoritma optičnega toka (glejte poglavje 5.3).

Na uporabniškem vmesniku smo dodali *VideoPreviewLayer*, zato je slika, ki jo vidimo, direktna neobdelana slika iz videokamere na napravi. iOS omogoča izbiro različnih ravni kakovosti zajetega posnetka. Izbrali smo nizkokakovostno (ang. *low quality*) zajemanje videa, saj tako pospešimo obdelavo, ker slik ni treba zmanjševati. Slika 6.7 prikazuje začetek postopka zaznave cestišča, kjer je viden osnovni uporabniški vmesnik načina v živo.

Aplikacija omogoča tudi način razhroščevanje oz. »debug mode«. V tem načinu aplikacija izrisuje vmesne rezultate posameznih algoritmov, ki tečejo v ozadju. Podrobnejše delovanje aplikacije bomo s pomočjo tega načina opisali



Slika 6.7: Osnovni uporabniški vmesnik načina v živo



Slika 6.8: Dodatni meni načina razhroščevanja

v nadaljevanju.

6.3 Skupna funkcionalnost – DebugMode

Aplikacija ima možnost razhroščevanja. V tem načinu se uporabniku na uporabniškem vmesniku prikaže dodaten meni, ki ga prikazuje Slika 6.8. Uporabnik med prikazom rezultatov podpostopkov preklaplja s pritiskom na ustrezni gumb. Nad slojem *VideoPreviewLayer* na uporabniškem vmesniku je dodatno polje za sliko (*ImageView*), na katerem se prikazuje pomanjšana slika, ki je trenutno v obdelavi.

Kot smo opisali v prejšnjem poglavju, se postopek obdelave najprej začne z detekcijo/segmentacijo cestišča. To se zgodi s pomočjo metode *watershed*, z detekcijo robov vozišča in izračunom presečišča robov (poglavje 5.1). Slika 6.9 prikazuje prvi del postopka in njegov prikaz v načinu *debug*. Poleg menija in sloja *VideoPreviewLayer*, ki sta opisana v prejšnjem odstavku, so v sredini na dodatnem polju za sliko (*ImageView*) vidne naslednje informacije:

- cesta bele barve, drugo je osvetljeno (takšna obarvanost je posledica dejstva, da smo vhodni sliki s pomočjo metode *add* dodali segmentirano cestišče, kar je prikazano na Sliki 5.4),
- mogoči kandidati za robove vozišča v modri barvi (na Sliki 6.9 so prekriti z izbranim levim in desnim robom večje debeline),
- izbrani levi rob v rdeči barvi in izbrani desni rob v zeleni barvi,
- petkotnik v zeleni barvi, ki je omejen z robovi cestišča (zelena barva kaže, da nevarnosti ni).

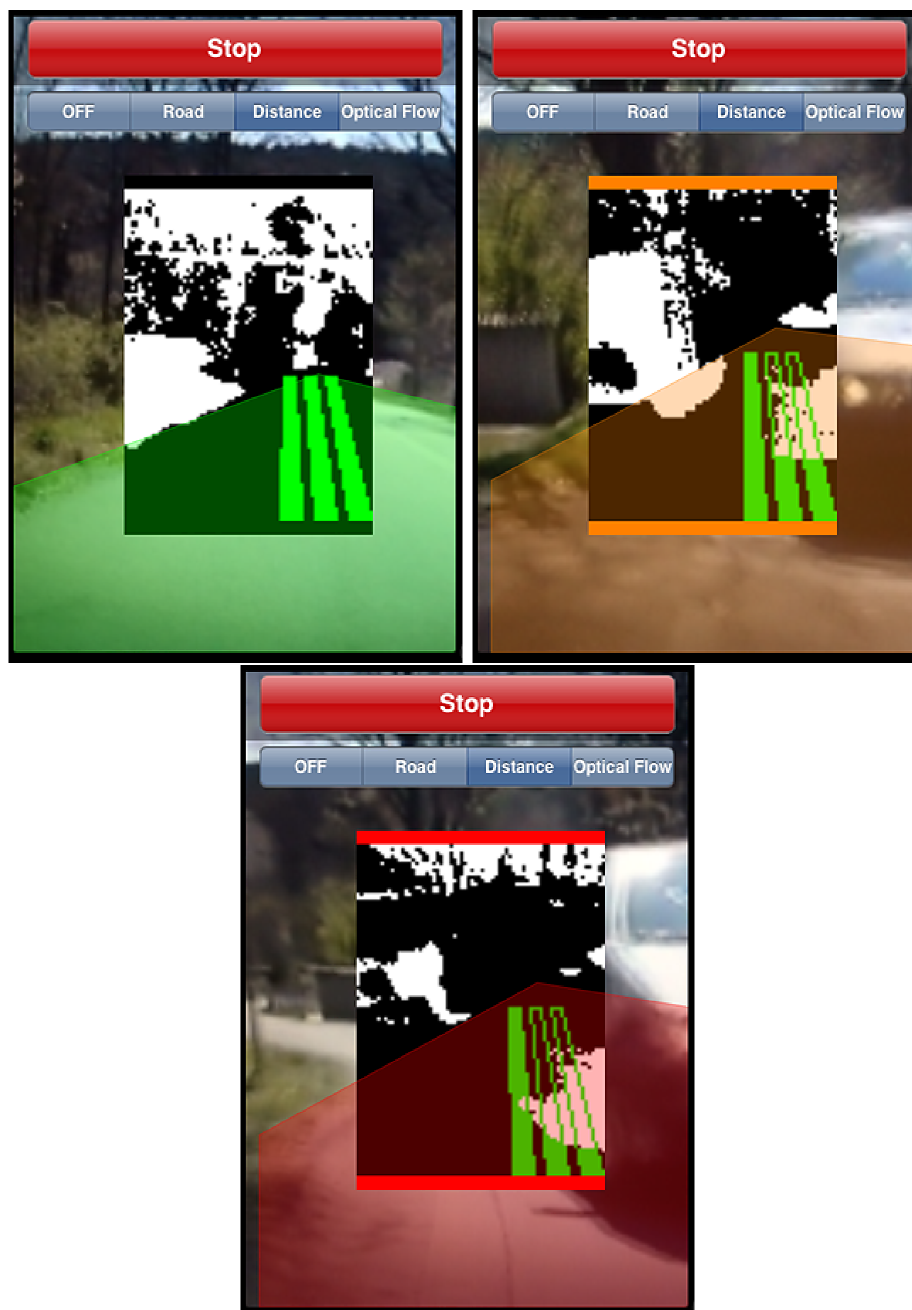
Naslednji korak v postopku in obenem naslednja možnost načina razhroščevanja je prikaz zaznavanja približevanja, kar opisuje poglavje 5.2. V tem primeru se v sredinsko sliko (ImageView) izrisuje binarna slika, pridobljena z enim izmed postopkov, ki so opisani v poglavju 5.2. Zgornji in spodnji rob slike sta obarvana v barvi trenutne stopnje nevarnosti. Enako je obarvan tudi petkotnik, ki označuje cestišče. Slika 6.10 prikazuje različne stopnje nevarnosti in posledično različne barve, ki označujejo stopnje nevarnosti. Različne stopnje nevarnosti ocenjujemo glede na razmerje med izmerjenim minimumom in razdaljo do presečišča, kar prikazuje Tabela 6.1.

Poleg vidnih opozoril na nevarnost s pomočjo različnih barv nas aplikacija na različne stopnje nevarnosti opozarja še zvočno, s piskanjem. Z naraščanjem stopnje nevarnosti se zmanjšuje interval med piski. Za lažjo predstavo lahko zvok, ki ga aplikacija predvaja, primerjamo s parkirnimi senzorji v avtomobilih.

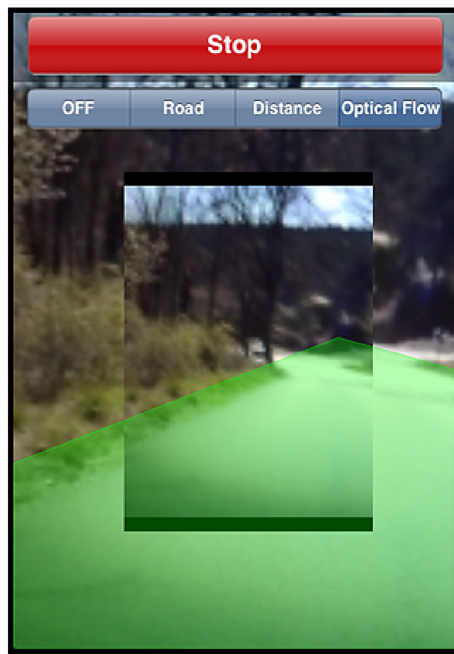
Slika 6.9: Prikaz delovanja detekcije cestišča v načinu razhroščevanja

stopnja nevarnosti	opozorilna barva	$\frac{minimum}{y_{presecisce}}$
nizka	zelena	0,92
srednja	oranžna	0,70
visoka	rdeča	0,25

Tabela 6.1: Razmerje med izmerjenim minimumom in razdaljo do presečišča, različne stopnje nevarnosti in barve, ki jih označujejo.



Slika 6.10: Barve za nevarnost približevanja, glede na stopnjo nevarnosti



Slika 6.11: Prikaz slike, ki se trenutno obdeluje z algoritmom optičnega toka.

Zadnja možnost prikaza je prikaz optičnega toka, ki ga uporabljamo za zaznavo prehitevanja in je opisan v poglavju 5.3. Ker je ta algoritem procesorsko požrešen, ga ne poganjamo na vsaki sličici, temveč le na vsaki četrti. Slika 6.11 prikazuje, kako se slika, ki jo trenutno obdelujemo, prikazuje nad slojem *VideoPreviewLayer*. Ko aplikacija zazna prehitevanje, izriše vektor, ki je zaznavo/nevarnost sprožil. Na zaznano nevarnost nas opozori z zvočnim piskom. Ta pisk se razlikuje od piska, ki opozarja na približevanje. Poleg tega se pisk, ki nas opozarja na prehitevanje, predvaja le v levem kanalu (če uporabljamo slušalke, se bo to predvajalo le v levem ušesu), kjer se prehitevanje tudi dogaja. Posledično aplikacija deluje pravilno le v državah, kjer se vozi po desni strani vozišča.

Poglavje 7

Analiza rezultatov

7.1 Uspešnost

Uspešnost postopka smo ocenjevali na 22 posnetkih. Vsi posnetki so nastali v popoldanskem času v podeželskem okolju. Prvih 19 posnetkov traja v povprečju 50 s, medtem ko zadnji trije trajajo v povprečju 200 s. Dva posnetka smo uporabili za gradnjo oz. učenje postopka. Ocenjevali smo uspešnost rešitve pri zaznavi obeh nevarnosti (približevanje in prehitevanje), opisanih v poglavjih 5.2 in 5.3. Tabela 7.1 prikazuje rezultate meritev na vzorčnih posnetkih, odebeljena posnetka sta učna posnetka.

Pravilnost delovanja postopka smo merili tako, da smo spremljali posnetke, medtem ko jih je aplikacija obdelovala. Označili smo, kdaj nas je aplikacija pravilno opozorila na nevarnost (TP), kdaj nas ni opozorila na nevarnost, ko je ta bila (FN), in kdaj nas je opozorila na nevarnost, ko te ni bilo (FP). Rezultati so v Tabeli 7.1.

Zaznave cestišča nismo natančneje obravnavali, saj je zaradi hitre spremenljivosti težko merljiva. Ker pa je ta zaznava bistvena/temeljna za druga dva postopka, lahko ocenimo, da je dokaj natančna in deluje zelo dobro, saj so končni rezultati dovolj kakovostni.

Za zaznavo približevanja in zaznavo prehitevanja smo izračunali pogojni verjetnosti s spodnjima enačbama.

Posnetek	**Prehitevanje**				**Približevanje**			
(št.)	N	TP	FP	FN	N	TP	FP	FN
1	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**
2	1	0	0	1	0	0	2	0
3	2	1	0	1	0	0	1	0
4	1	1	0	0	0	0	0	0
5	3	2	0	1	0	0	1	0
6	1	0	0	1	0	0	2	0
7	1	1	1	0	0	0	1	0
8	2	2	1	0	0	0	2	0
9	1	1	0	0	0	0	0	0
10	1	1	1	0	0	0	1	0
11	2	2	0	0	0	0	0	0
12	1	1	0	0	0	0	0	0
13	1	1	1	0	0	0	4	0
14	1	1	0	0	0	0	0	0
15	1	1	0	0	0	0	0	0
16	2	1	0	1	0	0	3	0
17	1	1	0	0	0	0	0	0
18	1	0	0	1	0	0	1	0
19	2	2	0	0	0	0	0	0
20	**3**	**3**	**2**	**0**	**5**	**4**	**1**	**1**
21	2	0	0	2	4	2	2	2
22	1	1	4	0	5	5	2	0
$\sum$	32	24	10	8	14	11	23	3

Tabela 7.1: Rezultati meritev na vzorčnih posnetkih, učna posnetka sta odebeljena (N – število nevarnosti v posnetku).

$$P(nevarnost|opozorilo) = \frac{TP}{TP + FP}$$
$$P(opozorilo|nevarnost) = \frac{TP}{TP + FN}$$

Za zaznavanje prehitevanja izračunamo:

$$P(nevarnost_{prehitevanje}|opozorilo) =$$
$$\frac{TP_{prehitevanje}}{TP_{prehitevanje} + FP_{prehitevanje}} = 0,71$$

in

$$P(opozorilo|nevarnost_{prehitevanje}) =$$
$$\frac{TP_{prehitevanje}}{TP_{prehitevanje} + FN_{prehitevanje}} = 0,75$$

Za zaznavanje približevanja izračunamo:

$$P(nevarnost_{priblizevanje}|opozorilo) =$$
$$\frac{TP_{priblizevanje}}{TP_{priblizevanje} + FP_{priblizevanje}} = 0,32$$

in

$$P(opozorilo|nevarnost_{priblizevanje}) =$$
$$\frac{TP_{priblizevanje}}{TP_{priblizevanje} + FN_{priblizevanje}} = 0,79$$

Aplikacija nas z verjetnostjo 0,75 opozori na nevarnost prehitevanja in z verjetnostjo 0,79 na nevarnost približevanja, če se nevarnosti zgodita. Če nas aplikacija opozori na nevarnost prehitevanja, je verjetnost, da je ta nevarnost resnična, 0,71. Če nas aplikacija opozori na nevarnost približevanja, je verjetnost, da je ta nevarnost resnična, 0,32.

Postopku zaznave prehitevanja povzročajo težave naslednje stvari.

1. Če cestišče še ni zaznano, se tudi postopek zaznave prehitevanja ne izvede. Na nekaterih posnetkih avtomobil pripelje takoj na začetku (preden je cestišče dovolj dobro zaznano), zato aplikacija ne zazna prehitevanja.

2. Ker postopek optičnega toka izvajamo redkeje, včasih ne zaznamo dveh bližnjih nevarnosti, npr. dveh avtomobilov, ki nas zaporedoma prehitita.

To sta dva izmed poglavitnih razlogov za napačno negativne rezultate (FN). Napačno pozitivne (FP) rezultate dobimo zaradi šuma v algoritmu zaznave optičnega toka. Ta šum smo poskušali minimizirati, vendar delno še vedno ostane, kar se pozna v napačnih meritvah. Drugi razlog za napačno pozitivne meritve pa je v visoki gibljivosti kolesa, zaradi katere nastanejo sunki, ki jih aplikacija interpretira/zazna kot nevarnosti prehitevanja.

Postopek zaznave približevanja kloni pred dvema težavama: sencami na cestišču in prezgodnjim opozarjanjem. Aplikacija interpretira sence kot ovire na cesti in nas opozori nanje kot na nevarnosti.

Ker želimo, da nas aplikacija opozori na nevarnosti, ko se nam avtomobil šele približuje, zaznavamo bližino avtomobila tudi, ko je ta še zelo daleč (Tabela 6.1 parameter za nizko stopnjo nevarnosti). Ker pa kolo ni stabilno in, kot že omenjeno, prihaja do sunkov, včasih v daljavi namesto avtomobila kot nevarnost zaznamo rob cestišča. To se zgodi, ker se majhen sunek oziroma sprememba gibanja na kolesarskem krmilu kaže kot velika sprememba na območju v bližini presečišča robov cestišča (v daljavi).

Včasih se zgodi, da zaznamo obe nevarnosti naenkrat, aplikacija pa nas opozori samo na eno. Ker smo postopka ocenjevali ločeno, smo šteli zaznavo ene nevarnosti namesto obeh kot napako (napačno negativno). Ta napaka se nam ne zdi bistvenega pomena, saj nas aplikacija, četudi z zaznavo le ene nevarnosti, opozori in tako dosega svoj namen.

Menimo, da je bolje, če nas postopek opozori na resnične nevarnosti in nas obenem opozori še na dodatne lažne, kot pa da nas ne opozori na katero izmed resnih nevarnosti.

7.2 Počasnost delovanja

Postopek smo najprej testirali na iOS Simulatorju. To je del razvojnega okolja Xcode, kjer se testirajo aplikacije iOS. iOS Simulator ima zavajajoče ime (simulator), saj se v njem aplikacija poganja na procesorju namiznega računalnika, ki je precej hitrejši kot procesor v napravi sami. Delovanje aplikacije je bilo po prenosu na napravo zato presenetljivo počasno.

Hitrost smo skušali izboljšati s pomanjšanjem velikosti obdelanih sličic na četrtino izvorne velikosti. Procesorsko zahtevne algoritme smo poskušali izvajati redkeje. Oba popravka hkrati sta omogočila sprejemljivo hitrost izvajanja, pri čemer je ozko grlo še vedno hitrost izvajanja metod OpenCV na ciljni napravi. Glede na rezultate, opisane v začetku poglavja, smo zaradi hitrosti delovanja žrtvovali določen del uspešnosti postopka.

Kot glavno merilo hitrosti delovanja smo vzeli število sličic, ki jih uspe naprava obdelati v sekundi (fps). Na opazovanih napravah (iPad 2 in iPad 3), ki sta sledeči si izvedbi istega modela naprave, so razlike precejšnje (Tabela 7.2). Za druge ciljne naprave lahko, glede na hitrost procesorja, upravičeno domnevamo, da bo število obdelanih sličic v sekundi za določene naprave večje, za druge pa manjše od opazovanih. Ker se ta tehnologija v tem trenutku zelo hitro razvija, lahko predvidevamo, da se bodo zmogljivosti v prihodnosti še izboljšale ter bomo lahko uporabili še druge natančnejše in procesorsko zahtevnejše postopke. Večina omejitev, ki smo jih imeli pri razvoju postopka, je namreč posredno in neposredno povezana s procesorjem, ki je vgrajen v napravah, oziroma njegovo hitrostjo.

	iPad 2	iPod Touch (5. gen.)	iPhone 4S	iPad 3 (new iPad)	iPhone 5
CPE (dvojedrni ARM)	Cortex-A9	Cortex-A9	Cortex-A9	Cortex-A9	v7 lastne zasnove
frekvenca CPE (MHz)	1000	1000 (zmanjšana na 800)		1000	1200
RAM (MB)	512	256	512	1024	1024
GPE	SGX543MP2	SGX543MP2	SGX543MP2	SGX543MP4	SGX543MP3
fps aplikacije	**≈ 3**			**≈ 7**	
fps aplikacije s četrtino loč.	**≈ 7**			**≈ 18**	
ocenjen fps		3 do 7	3 do 7		>18

Tabela 7.2: Specifikacije novejših naprav. Za opazovani napravi iPad 2 in 3 je navedeno tudi število obdelanih sličic na sekundo (fps) med izvajanjem aplikacije pred izboljšavo in po njej (zmanjšanje velikosti sličic na četrtino). Za druge naprave je podana ocena fps, ocenjena na podlagi primerjave specifikacij.

Knjižnica OpenCV ima določene metode optimizirane tako, da lahko na namiznih računalnikih izkoriščajo tudi grafične procesne enote. Za mobilne naprave te optimizacije še ne obstajajo, vendar pričakujemo, da v prihodnosti bodo (npr. knjižnica FastCV [25]). Postopek bi se dalo verjetno precej optimizirati z uporabo senčilnikov in ukazov OpenCL [28], ki nam bi omogočali, da del računanja prenesemo na grafično procesno enoto (GPE) v mobilni napravi. GPE v mobilnih napravah se namreč izboljšujejo ravno tako skokovito kot procesorji in so v našem postopku še neuporabljene.

7.3 Poraba baterije

Merjenje porabe baterije smo izvedli tako, da smo aplikacijo 10 minut poganjali na opazovani napravi. Meritev smo večkrat ponovili in izračunali povprečno porabo. Ugotovili smo, da aplikacija porabi odstotek baterije za vsako minuto delovanja. Iz tega je razvidno, da je aplikacija, kljub našim prizadevanjem za optimizacijo, še vedno energijsko potratna. Predviden čas delovanja naprav prikazuje Tabela 7.3 [21]. Kot izhodišče za oceno neizmerjenih vrednosti smo uporabili proizvajalčevo napoved trajanja baterije. Napoved ne ustreza vedno dejanskim časom delovanja, vendar je to edino enotno merilo za primerjavo med napravami, ki ga imamo na voljo. Za izračun ocene časa trajanja naprav smo uporabili naslednjo enačbo:

$$t_{naprava} = \frac{t_{meritev} * napoved_{naprava}}{napoved_{meritev}}$$

V Tabeli 7.3 so navedene tudi kapacitete baterij naprav. Te se med seboj precej razlikujejo, razmerja med kapaciteto in napovedjo trajanja baterije so namreč nesorazmerna. Prav zaradi te nesorazmernosti smo za oceno predvidenega časa delovanja naprav namesto kapacitete baterije uporabili proizvajalčevo napoved.

Model naprave	**Proizvajalčeva napoved (h)**	**Kapaciteta baterije (mAh)**	**Pričakovani čas delovanja aplikacije (min)**
iPad 2	**10**	6944	**100**
iPod Touch	6	930	≈60
iPhone 4S	9	1430	≈90
iPad 3	10	11666	≈100
iPhone 5	10	1440	≈100

Tabela 7.3: Čas delovanja aplikacije na opazovani napravi in ocenjene vrednosti za druge naprave

Poglavje 8

Sklepne ugotovitve

Kot se je izkazalo pri testiranju, ima postopek opozarjanja kolesarja na nevarnosti približevanja in prehitevanja, ki smo ga razvili, pomanjkljivosti. Nedvomno obstajajo možnosti za izboljšave. Po drugi strani pa postopek lahko služi kot uporabna podlaga za nadaljnji razvoj.

Postopek je razdeljen na tri podpostopke: segmentacijo ceste in določitev njenih robov, zaznavanje približevanja in zaznavanje prehitevanja. Kljub našim prizadevanjem za optimizacijo je aplikacija še vedno energijsko potratna in v najboljšem primeru deluje le 100 minut. Za njeno sprejemljivo delovanje (med 7 in 18 fps) smo žrtvovali tudi del uspešnosti postopka. Aplikacija nas opozori na 75–80 % nevarnosti, vendar je lažnih opozoril okoli 30–70 %.

Mogoče izboljšave delovanja aplikacije vidimo na več področjih. Delovanje postopka lahko izboljšamo z izbiro postopkov zaznave, ki so odporni na sence. Šum lahko odstranimo s pomočjo dodatnih senzorjev v napravi, ki jih trenutno ne izkoriščamo, npr. s pospeškometerom in žiroskopom. Aplikacija se s pomočjo distribucije v trgovini Apple App Store teoretično lahko namesti na sto ali več naprav. To funkcionalnost bi lahko uporabili za pridobivanje povratnih informacij o delovanju in dodatnih učnih podatkov. Tako bi s pomočjo strojnega učenja izboljšali postopek zaznave.

S pomočjo ukazov OpenCL bi lahko del obdelovanja podatkov dodelili

grafičnemu procesorju. Nekatere pretvorbe bi lahko pohitrili z uporabo zbirnih ukazov za procesor ARM (ukazi NANO). Pričakujemo lahko, da bodo v prihodnosti procesorji hitrejši in bo naš postopek na novejših napravah deloval hitreje.

Funkcionalne nadgradnje aplikacije vidimo v izboljšani uporabniški izkušnji (npr. prostoročno upravljanje aplikacije). Aplikaciji lahko dodamo možnost zaznave vozne strani in ji s tem omogočimo delovanje povsod, saj trenutno deluje le, če vozimo po desni strani.

Literatura

[1] J. M. Álvarez, A. M. López, and R. Baldrich. Shadow resistant road segmentation from a mobile monocular system. In *Proceedings of the 3rd Iberian conference on Pattern Recognition and Image Analysis, Part II*, IbPRIA '07, pages 9–16, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] M. Bertozzi, A. Broggi, G. Conte, and A. Fascioli. The experience of the argo autonomous vehicle. In *IN PROCS. SPIE'98 - AEROSENSE CONF*, pages 218–229, 1998.

[3] S. Beucher and Centre De Morphologie Mathmatique. The watershed transformation applied to image segmentation. In *Scanning Microscopy International*, pages 299–314, 1991.

[4] S. Beucher and Centre De Morphologie Mathmatique. Road segmentation and obstacle detection by a fast watershed transformation. In *Intelligent Vehicles '94 Symposium*, pages 296–301, 1994.

[5] G. R. Bradski and A. Kaehler. *Learning opencv, 1st edition.* O'Reilly Media, Inc., first edition, 2008.

[6] A. Broggi and S. Bertè. Vision-based road detection in automotive systems: A real-time expectation-driven approach. *Journal of Artificial Intelligence Research*, 3:325–348, 1995.

[7] Beucher Bilodeau Centre, S. Beucher, M. Bilodeau, and X. Yu. Road segmentation by watersheds algorithms. In *Proc. of PROMETHEUS workshop, Sophia-Antipolis*, 1990.

[8] H. Dahlkamp, A. Kaehler, D. Stavens, S. Thrun, and G. Bradski. Self-supervised monocular road detection in desert terrain. In *Proceedings of Robotics: Science and Systems*, Philadelphia, USA, August 2006.

[9] J. Debayle, A. Raihane, A. Belhaoua, O. Bonnefoy, G. Thomas, J. Chaix, and J. Pinoli. Velocity field computation in vibrated granular media using an optical flow based multiscale image analysis method. *Image Analysis and Stereology*, 28(1), 2011.

[10] E. D. Dickmanns. *Dynamic Vision for Perception and Control of Motion*. Springer, 2007.

[11] N. Funk. A study of the kalman filter applied to visual tracking. Technical report, University of Alberta, 2003.

[12] H. Kong, J. Audibert, and J. Ponce. Vanishing point detection for road detection.

[13] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke, D. Johnston, S. Klumpp, D. Langer, A. Levandowski, J. Levinson, J. Marcil, D. Orenstein, J. Paefgen, I. Penny, A. Petrovskaya, M. Pflueger, G. Stanek, D. Stavens, A. Vogt, and S. Thrun. Junior: The stanford entry in the urban challenge. *Journal of Field Robotics*, 2008.

[14] F. Ren, J. Huang, M. Terauchi, R. Jiang, and R. Klette. Lane detection on the iphone. In *Arts and Technology. First International Conference, ArtsIT 2009, Yi-Lan, Taiwan, September 24-25, 2009, Revised Selected Papers*, 5 2012.

[15] J. Shi and C. Tomasi. Good features to track. In *1994 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94)*, pages 593–600, 1994.

[16] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley,

M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney. Winning the darpa grand challenge. *Journal of Field Robotics*, 2006.

[17] C. Urmson, J. Anhalt, J. A. Bagnell, C. R. Baker, R. E. Bittner, J. M. Dolan, D. Duggins, D. Ferguson, T. Galatali, H. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. Howard, A. Kelly, D. Kohanbash, M. Likhachev, N. Miller, K. Peterson, R. Rajkumar, P. Rybski, B. Salesky, S. Scherer, Y. Seo, R. Simmons, S. Singh, J. M. Snider, A. Stentz, W. L. Whittaker, and J. Ziglar. Tartan racing: A multi-modal approach to the darpa urban challenge. Technical Report CMU-RI-TR-, Robotics Institute, http://archive.darpa.mil/grandchallenge/, April 2007.

[18] C. Urmson, C. R. Baker, J. M. Dolan, P. Rybski, B. Salesky, W. L. Whittaker, D. Ferguson, and M. Darms. Autonomous driving in traffic: Boss and the urban challenge. *AI Magazine*, 30(2):17–29, June 2009.

[19] A. Vadivel, M. Mohan, S. Sural, and A. K. Majumdar. Segmentation using saturation thresholding and its application in content-based retrieval of images. In *Lecture Notes in Computer Science*, pages 33–40, 2004.

[20] G. Welch and G. Bishop. An introduction to the kalman filter. Technical report, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1995.

[21] Compare iPad Models.
`http://www.apple.com/ipad/compare/`, 2012.

[22] Computer Vision Talks.
`http://computer-vision-talks.com/`, 2011.

[23] Computer vision with iOS Part 1: Building an OpenCV framework.
http://aptogo.co.uk/2011/09/opencv-framework-for-ios/, 2011.

[24] EMGU CV.
http://www.emgu.com, 2012.

[25] FastCV.
https://developer.qualcomm.com/mobile-development/mobile-technologies/computer-vision-fastcv, 2012.

[26] iOnRoad.
http://www.ionroad.com, 2012.

[27] Nevada Department of Motor Vehicles.
http://www.dmvnv.com/news/12005-autonomous-vehicle-licensed.htm, 2012.

[28] OpenCL - The open standard for parallel programming of heterogeneous systems.
http://www.khronos.org/opencl/, 2012.

[29] OpenCV - Open Source Computer Vision Library.
http://opencv.org/, 2012.

[30] Stanley.
http://cs.stanford.edu/group/roadrunner/stanley.html, 2006.

[31] The ARGO Project.
http://www.argo.ce.unipr.it, 1999.